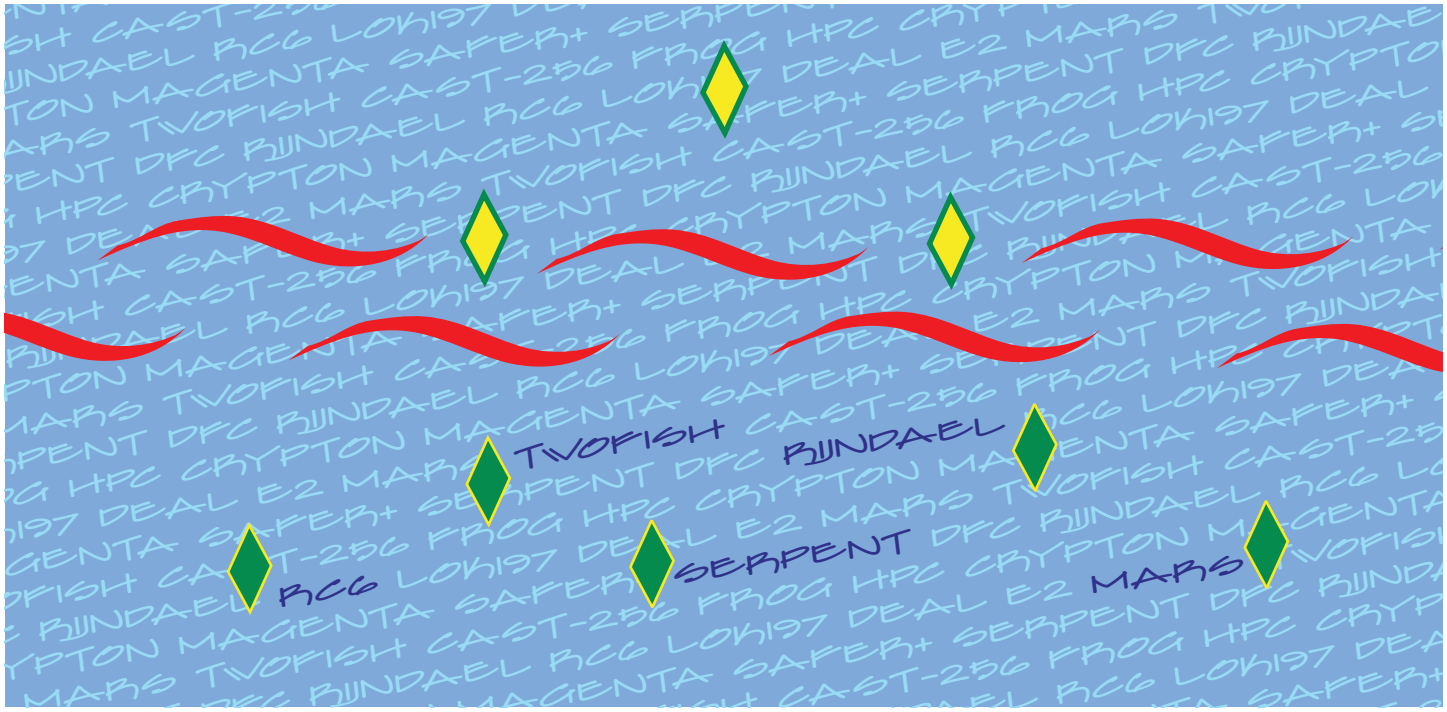
AES

A Crypto Algorithm for the Twenty-first Century . . .



The Third Advanced Encryption Standard Candidate Conference

APRIL 13-14, 2000

HILTON NEW YORK AND TOWERS
New York, NY, USA

http://www.nist.gov/aes

NIST
National Institute of Standards and Technology
Technology Administration, U.S. Department of Commerce

Preface

The Third Advanced Encryption Standard Candidate Conference (AES3) is the last in a series of three conferences that NIST has organized in its quest to develop the AES. It has been a long road, since NIST first announced its intention in January 1997 to develop a replacement standard for DES. Now, AES3 presents a wonderful opportunity for the cryptographic community to gather and discuss Round 2 analysis and other issues that are critical to the AES development effort. After Round 2 ends on May 15, 2000, NIST will begin the process of selecting the algorithm(s) that will be included in a draft AES Federal Information Processing Standard (FIPS). Therefore, NIST is holding AES3 to better understand which of the finalist algorithms - MARS, RC6™, Rijndael, Serpent, and Twofish - should be selected for the FIPS.

The papers to be presented at AES3 cover a wide range of issues, including cryptanalysis, implementability in Field Programmable Gate Arrays (FPGAs), hardware simulations, performance on various platforms, the role of future resiliency, and the possibility of including single or multiple algorithms in the AES FIPS.

Please see the AES home page at http://www.nist.gov/aes for the remaining papers that were proposed for AES3. Those papers - like the ones presented at AES3 - are considered official Round 2 public comments.

All Round 2 official public comments are due by May 15, 2000, and they should be submitted to AESRound2@nist.gov. This also includes any comments that interested parties may have on the papers presented at both AES3 and FSE 2000 (e.g., comments on their validity, and their applicability to and impact on the AES selection). NIST is eager to hear responses to these results and research.

The Program Committee members deserve a lot of credit for their hard work in evaluating papers, preparing for the conference, and chairing the panel presentations: Miles Smid (CygnaCom Solutions), Morris Dworkin (NIST), Tom Berson (Anagram Laboratories), Dennis Branstad (consultant, TIS Labs), Craig Clapp (PictureTel), Susan Langford (Certicom Corp.), Stefan Lucks (Universität Mannheim), Tim Moses (Entrust Technologies), and David Solo (Citigroup).

Special thanks go to the NIST staff who have provided invaluable assistance in evaluating documents and planning for AES3: Elaine Barker, Larry Bassham, Bill Burr, Jim Dray, Morris Dworkin, Jim Nechvatal, Ed Roback, and Juan Soto. Much gratitude is extended to the NIST staff responsible for the logistical side of AES3: Kathy Kilmer, Lori Phillips, and Vickie Harris.

A special mention of thanks must be made for the cooperation and assistance provided by Bruce Schneier, chair of the FSE 2000 Program Committee, and Beth Friedman of Counterpane Labs, for their efforts to coordinate these two conferences.

Finally - and most importantly - NIST greatly appreciates the efforts of all the authors who submitted papers for AES3. We have said this before, and we will say it again: the ultimate success of the AES Development Effort depends heavily on the public evaluation and analysis performed by the cryptographic community. Thank you for your hard work.

Personally, I would like to thank Miles Smid for his tireless leadership role in the AES development effort over the years, laying the solid foundation needed to support any future success that may be enjoyed by the AES.

We hope that you benefit a great deal from having joined us in New York City.

Jim Foti
NIST

April 2000

Third Advanced Encryption Standard Candidate Conference: AES3

Table of Contents

Session 3: "Surveys"

Session 4: "Cryptographic Analysis and Properties" (I)

*****

Day 2 - Friday, April 14, 2000

Session 5: "Cryptographic Analysis and Properties" (II)

Session 6: "AES Issues" Panel

Session 7: "ASIC Evaluations / Individual Algorithm Testing"

Abstracts of AES-related Papers from the Fast Software Encryption Workshop (FSE) 2000

Bruce Schneier
Chair, FSE 2000 Program Committee

The Seventh Fast Software Encryption Workshop (FSE 2000) was held during the three days immediately before this AES conference. Seven papers related to the AES finalists were presented at FSE 2000, and the titles and abstracts for those papers are listed below.

The proceedings for FSE 2000 will be published by Springer-Verlag in their Lecture Notes in Computer Science series. Copies of the pre-proceedings are available from the FSE secretariat.

***

Title: *Improved Cryptanalysis of Rijndael*
Authors: Niels Ferguson, John Kelsey, Bruce Schneier, Mike Stay, David Wagner, and Doug Whiting
Abstract: We improve the best attack on 6-round Rijndael from complexity 2^{72} to 2^{42}. We also present the first known attacks on 7- and 8-round Rijndael. Finally, we discuss the key schedule of Rijndael and describe a related-key technique that can break 9-round Rijndael with 256-bit keys.

Title: *On the Pseudorandomness of AES Finalists -- RC6, Serpent, MARS and Twofish*
Authors: Tetsu Iwata and Kaoru Kurosawa
Abstract: The aim of this paper is to compare the security of AES finalists in an idealized model like Luby and Rackoff. We mainly prove that a five round idealized RC6 and a three round idealized Serpent are super-pseudorandom permutations. We then show a comparison about this kind of pseudorandomness for four AES finalists, RC6, Serpent, MARS and Twofish.

Title: *Correlations in RC6*
Authors: Lars Knudsen and Willi Meier
Abstract: In this paper the block cipher RC6 is analysed. RC6 is submitted as a candidate for the Advanced Encryption Standard, and is one of five finalists. It has 128-bit blocks and supports keys of 128, 192 and 256 bits, and is an iterated 20-round block cipher. Here it is shown that versions of RC6 with 128-bit blocks can be distinguished from a random permutation with up to 15 rounds; for some weak keys up to 17 rounds. Moreover, with an increased effort key-recovery attacks can be mounted on RC6 with up to 15 rounds faster than an exhaustive search for the key.

Title: *Securing the AES Finalists Against Power Analysis Attacks*
Author: Thomas Messerges
Abstract: Techniques to protect software implementations of the AES candidate algorithms from power analysis attacks are investigated. New countermeasures that employ random masks are developed and the performance characteristics of these countermeasures are analyzed. Implementations in a 32-bit, ARM-based smartcard are considered.

Title: *Efficient Methods for Generating MARS-like S-boxes*
Authors: L. Burnett, G. Carter, E. Dawson, and W. Millan
Abstract: One of the five AES finalists, MARS, makes use of a 9x32 s-box with very specific combinatorial, differential and linear correlation properties. The s-box used in the cipher was selected as the best from a large sample of pseudo randomly generated tables, in a process that took IBM about a week to compute. This paper provides a faster and more effective alternative generation method using heuristic techniques to produce 9x32 s-boxes with cryptographic properties that are clearly superior to those of the MARS s-box, and typically take less than two hours to produce on a single PC.

Title: *A Statistical Attack on RC6*
Authors: Henri Gilbert, Helena Handschuh, Antoine Joux, and Serge Vaudenay
Abstract: This paper details the attack on RC6 which was announced in a report published in the proceedings of the second AES candidate conference (March 1999). Based on an observation on the RC6 statistics, we show how to distinguish RC6 from a random permutation and to recover the secret extended key for a fair number of rounds.

Title: *Amplified Boomerang Attacks Against Reduced-Round MARS and Serpent*
Authors: John Kelsey, Tadayoshi Kohno, and Bruce Schneier
Abstract: We introduce a new kind of attack based on Wagner's boomerang and inside-out attacks. We first describe the new attack in terms of the original boomerang attack, and then demonstrate its use on reduced-round variants of the MARS core and of Serpent. Our attack breaks eleven rounds of the Mars core with 2^{65} chosen plaintexts, 2^{69} memory, and 2^{229} partial decryptions. Our attack breaks eight rounds of Serpent with 2^{114} chosen plaintexts, 2^{119} memory, and 2^{179} partial decryptions.

Session 1:

"FPGA Evaluations"

An FPGA Implementation and Performance Evaluation of the AES Block Cipher Candidate Algorithm Finalists *

AJ Elbirt[1], W Yip[1], B Chetwynd[2], C Paar[1]
Electrical and Computer Engineering Department
Worcester Polytechnic Institute
100 Institute Road, Worcester, MA 01609, USA

[1] Email: {aelbirt, waihyip, christof}@ece.wpi.edu
[2] Email: spunge@alum.wpi.edu

Abstract

The technical analysis used in determining which of the Advanced Encryption Standard candidates will be selected as the Advanced Encryption Algorithm includes efficiency testing of both hardware and software implementations of candidate algorithms. Reprogrammable devices such as Field Programmable Gate Arrays (FPGAs) are highly attractive options for hardware implementations of encryption algorithms as they provide cryptographic algorithm agility, physical security, and potentially much higher performance than software solutions. This contribution investigates the significance of FPGA implementations of four of the Advanced Encryption Standard candidate algorithm finalists. Multiple architectural implementation options are explored for each algorithm. A strong focus is placed on high throughput implementations, which are required to support security for current and future high bandwidth applications. The implementations of each algorithm will be compared in an effort to determine the most suitable candidate for hardware implementation within commercially available FPGAs.

Keywords: cryptography, algorithm-agility, FPGA, block cipher, VHDL

1 Introduction

The National Institute of Standards and Technology (NIST) has initiated a process to develop a Federal Information Processing Standard (FIPS) for the Advanced Encryption Standard (AES), specifying an Advanced Encryption Algorithm to replace the Data Encryption Standard (DES) which expired in 1998 [1]. NIST has solicited candidate algorithms for inclusion in AES, resulting in fifteen official candidate algorithms of which five have been selected as finalists. Unlike DES, which was designed specifically for hardware implementations, one of the design criteria for AES candidate algorithms is that they can be efficiently implemented in both hardware and software. Thus, NIST has announced that both hardware and software performance measurements will be included in their efficiency testing. So far, however, virtually all performance comparisons have been restricted to software implementations on various platforms [2].

The advantages of a software implementation include ease of use, ease of upgrade, portability, and flexibility. However, a software implementation offers only limited physical security, especially with respect to key storage [3] [4]. Conversely, cryptographic algorithms (and their associated keys) that are implemented in hardware are, by nature, more physically secure as they cannot easily be read or modified by an outside

*This research was supported in part through NSF CAREER award #CCR-9733246.

attacker [4]. The downside of traditional (ASIC) hardware implementation are the lack of flexibility with respect to algorithm and parameter switch. A promising alternative for implementation block cipher are reconfigurable hardware devices such as Field Programmable Gate Arrays (FPGAs). FPGAs are hardware devices whose function is not fixed and which can be programmed in-system. The potential advantages of encryption algorithms implemented in FPGAs include:

Algorithm Agility This term refers to the switching of cryptographic algorithms during operation. The majority of modern security protocols, such as SSL or IPsec, allow for multiple encryption algorithms. The encryption algorithm is negotiated on a per-session basis; e.g., IPsec allows among others DES, 3DES, Blowfish, CAST, IDEA, RC4 and RC6 as algorithms, and future extensions are possible. Whereas algorithm agility is costly with traditional hardware, FPGAs can be reprogrammed on-the-fly.

Algorithm Upload It is perceivable that fielded devices are upgraded with a new encryption algorithm which did not exist (or was not standardized!) at design time. In particular, it is very attractive for numerous security products to be upgraded for use of AES once the selection process is over. Assuming there is some kind of (temporary) connection to a network such as the Internet, FPGA-equipped encryption devices can upload the new configuration code.

Algorithm Modification There are applications which require modification of a standardized algorithm, e.g., by using proprietary S-boxes or permutations. Such modifications are easily made with reconfigurable hardware. Similarly, a standardized algorithm can be swapped with a proprietary one. Also, modes of operation can be easily changed.

Architecture Efficiency In certain cases, a hardware architecture can be much more more efficient if it is designed for a specific set of parameters; e.g., constant multiplication (of integers or in Galois fields) is far more efficient than general multiplication. With FPGAs it is possible to design and optimize an architecture for a specific parameter set.

Throughput Although typically slower than an ASIC implementations, FPGA implementations have the potential of running substantially faster then software implementations.

Cost Efficiency The time and costs for developing an FPGA implementation of a given algorithm are much lower than for an ASIC implementation. (However, for high-volume applications, ASIC solutions usually become the more cost-efficient choice.)

Note that algorithm agility remains an open research issue in regards to speed, physical security, and the cost associated with current high-end FPGA devices. However, we believe that cost is not a long-term limiting factor, as will be discussed in Section 3.3. For these reasons, this paper describes a thorough comparison the AES finalist algorithms RC6, Rijndael, Serpent, and Twofish with respect to implementation on state-of-the-art FPGAs. One aspect that seems to be especially relevant is the investigation of achievable encryption rates for FPGA-based implementations. We demonstrate that FPGA solutions encrypt at rates in the Gigabit range for all four algorithms investigated, which is at least one order of magnitude faster than most reported software implementations [5].

What follows is an investigation of the AES finalists to determine the nature of their underlying components. The characterization of the algorithms' components will lead to a discussion of the hardware architectures best suited for implementation of the AES finalists. A performance metric to measure the hardware cost for the throughput achieved by each algorithm's implementations will be developed and a target FPGA will be chosen so as to yield implementations that are optimized for high-throughput operation within the commercially available device. Finally, multiple architecture options of the algorithms within the targeted FPGA will be discussed and the overall performance of the implementations will be evaluated versus typical software implementations.

2 Previous Work

As opposed to custom hardware or software implementations, little work exists in the area of block cipher implementations within existing FPGAs. DES, the most common block cipher implementation targeted to FPGAs, has been shown to operate at speeds of up to 400 Mbit/s [6]. We believe that this performance can be greatly enhanced using today's technology. These speeds are significantly faster than the best software implementations of DES [7] [8] [9], which typically have throughputs below 100 Mbit/s, although a 137 Mbit/s implementation has been reported as well [7]. This performance differential is an expected result of DES having been designed in the 1970s with hardware implementations in mind.

Other block ciphers have been implemented in FPGAs with varying degrees of success. A typical example is the IDEA block cipher which has been implemented at speeds ranging from 2.8 Mbit/s [10] to 528 Mbit/s [11]. Note that while the 528 Mbit/s throughput was achieved in a fully pipelined architecture, the implementation required four Xilinx XC4000 FPGAs.

Some FPGA implementation throughputs for the AES candidates have been shown to be far slower than their software counterparts. Hardware throughputs of about 12 Mbit/s [12] [13] have been achieved for CAST-256. However, software implementations have resulted in throughputs of 37.8 Mbit/s for CAST-256 on a 200 MHz PentiumPro PC [5], a factor of three faster than FPGA implementations. When scaled to a more current 600 MHz PentiumPro PC, it is expected that the same software implementation would outperform FPGA implementations by an even larger factor. While an FPGA implementation of RC6 achieved data rates of 37.8 Mbit/s [13], our findings indicate that considerably higher data rates are achievable.

When examining the AES finalists, it is important to note that they do not necessarily exhibit similar behavior to DES when comparing hardware and software implementations. One reason for this is that the AES finalists have been designed with efficient software implementations in mind. Additionally, software implementations may be executed on processors operating at frequencies as high as 800 MHz while typical implementations that target FPGAs reach a maximum clock frequency of 50 MHz.

3 Methodology

3.1 Design Methodology

There are two basic hardware design methodologies currently available: language based (high level) design and schematic based (low level) design. Language based design relies upon synthesis tools to implement the desired hardware. While synthesis tools continue to improve, they rarely achieve the most optimized implementation in terms of both area and speed when compared to a schematic implementation. As a result, synthesized designs tend to be (slightly) larger and slower than their schematic based counterparts. Additionally, implementation results can greatly vary depending on the synthesis tool as well as the design being synthesized, leading to potentially increased variances in the synthesized results when comparing synthesis tool outputs. This situation is not entirely different from a software implementation of an algorithm in a high-level language such as C, which is also dependent on coding style and compiler quality. As shown in [14], schematic based design methodologies are no longer feasible for supporting the increase in architectural complexity evidenced by modern FPGAs. As a result, a language based design methodology was chosen as the implementation form for the AES finalists with VHDL being the specific language chosen.

3.2 Implementations — General Considerations

Each AES finalist was implemented in VHDL using a bottom-up design and test methodology. The same hardware interface was used for each of the implementations. In an effort to achieve the maximum efficiency possible, note that key scheduling and decryption were not implemented for each of the AES finalists. Because FPGAs may be reconfigured in-system, the FPGA may be configured for key scheduling and then later

reconfigured for either encryption or decryption. This option is a major advantage of FPGAs implementations over classical ASIC implementations. Round keys for encryption are loaded from the external key bus and are stored in internal registers and all keys must be loaded before encryption may begin. Key loading is disabled until encryption is completed. Each implementation was simulated for functional correctness using the test vectors provided in the AES submission package [15] [16] [17] [18]. After verifying the functionality of the implementations, the VHDL code was synthesized, placed and routed, and re-simulated with annotated timing using the same test vectors, verifying that the implementations were successful.

3.3 Selection of a Target FPGA

When examining the AES finalists for hardware implementation within an FPGA, a number of key aspects emerge. First, it is obvious that the implementation will require a large amount of I/O pins to fully support the 128-bit data stream at high speeds where bus multiplexing is not an option. It is desirable to decouple the 128-bit input and output data streams to allow for a fully pipelined architecture. Since the round keys cannot change during the encryption process, they may be loaded via a separate key input bus prior to the start of encryption. Additionally, to implement a fully pipelined architecture requires 128-bit wide pipeline stages, resulting in the need for a register-rich architecture to achieve a fast, synchronous implementation. Moreover, it is desirable to have as many register bits as possible per each of the FPGA's configurable units to allow for a regular layout of design elements as well as to minimize the routing required between configurable units. Finally, it is critical that fast carry-chaining be provided between the FPGA's configurable units to maximize the performance of AES finalists that utilize arithmetic operations [13] [12].

In addition to architectural requirements, scalability and cost must be considered. We believe that the chosen FPGA should be the best chip available, capable of providing the largest amount of hardware resources as well as being highly flexible so as to yield optimal performance. Unfortunately, the cost associated with current high-end FPGAs is relatively high (several hundred US dollars per device). However, it is important to note that the FPGA market has historically evolved at an extremely rapid pace, with larger and faster devices being released to industry at a constant rate. This evolution has resulted in FPGA cost-curves that decrease sharply over relatively short periods of time. Hence, selecting a high-end device provides the closest model for the typical FPGA that will be available over the expected lifespan of AES.

Based on the aforementioned considerations, the Xilinx Virtex XCV1000BG560-4 FPGA was chosen as the target device. The XCV1000 has 128K bits of embedded RAM divided among thirty-two RAM blocks that are separate from the main body of the FPGA. The 560-pin ball grid array package provides 512 usable I/O pins. The XCV1000 is comprised of a 64×96 array of look-up-table based Configurable Logic Blocks (CLBs), each of which acts as a 4-bit element comprised of two 2-bit slices for a total of 12288 CLB slices [19]. This type of configuration results in a highly flexible architecture that will accommodate the round functions' use of wide operand functions. Note that the XCV1000 also appears to be a good representative for a modern FPGA and that devices from other vendors are not fundamentally different. It is thus hoped that our results carry over, within limits, to other devices.

3.4 Design Tools

FPGA Express by Synopsys, Inc. and Synplify by Synplicity, Inc. were used to synthesize the VHDL implementations of the AES finalists. As this study places a strong focus on high throughput implementations, the synthesis tools were set to optimize for speed. As will be discussed in Section 6, the resultant implementations exhibit the best possible throughputs with the associated cost being an increase in the area required in the FPGA for each of the implementations. Similarly, if the synthesis tools were set to optimize for area, the resultant implementations would exhibit reduced area requirements at the cost of decreased throughput.

XACTstep 2.1i by Xilinx, Inc. was used to place and route the synthesized implementations. For the sub-pipelined architectures, a 40 MHz timing constraint was used in both the synthesis and place-and-route processes as it resulted in significantly higher system clock frequencies. However, the 40 MHz timing

constraint was found to have little affect on the other architecture types, resulting in nearly identical system clock frequencies to those achieved without the timing constraint.

Finally, Speedwave by Viewlogic Systems, Inc. and Active-HDL^{TM} by ALDEC, Inc. were used to perform behavioral and timing simulations for the implementations of the AES finalists. The simulations verified both the functionality and the ability to operate at the designated clock frequencies for the implementations.

4 Architecture Options and the AES Finalists

Before attempting to implement the AES finalists in hardware, it is important to understand the nature of each algorithm as well as the hardware architectures most suited for their implementation. What follows is an investigation into the key components of the AES finalists. Based on this breakdown, a discussion is presented on the hardware architectures most suited for implementation of the AES finalists.

4.1 Core Operations of the AES Finalist Algorithms

Algorithm	**XOR**	**Mod 2^{32} Add**	**Mod 2^{32} Subtract**	**Fixed Shift**	**Variable Rotate**	**Mod 2^{32} Multiply**	**GF(2^8) Multiply**	**LUT**
MARS	•	•	•	•	•	•		•
RC6	•	•		•	•	•		
Rijndael	•			•			•	•
Serpent	•			•				•
Twofish	•	•		•			•	•

Table 1: AES finalists core operations [20]

Modern FPGAs have a structure comprised of a two-dimensional array of configurable function units interconnected via horizontal and vertical routing channels. Configurable function units are typically comprised of look-up-tables and flip-flops. Look-up-tables may be configured as either combinational logic or memory elements. Additionally, many modern FPGAs provide variable-size SRAM blocks that may be used as either memory elements or look-up-tables [21].

In terms of complexity, the operations detailed in Table 1 that require the most hardware resources as well as computation time are the modulo 2^{32} multiplication and the variable rotation operations [20]. Implementing wide multipliers in hardware is an inherently difficult task that requires significant hardware resources. Additionally, algorithms that employ large variable rotations require a moderate amount of multiplexing hardware if carefully designed (see Section 5.1 for further discussion). S-Boxes may be implemented in either combinatorial logic or embedded RAM — the advantages of each of these options are discussed in Section 4.2. Fast operations such as bit-wise XOR, modulo 2^{32} addition and subtraction, and fixed value shifting are constructed from simple hardware elements. Additionally, the Galois field multiplications required in Rijndael and Twofish can also be implemented very efficiently in hardware as they are multiplications by a constant. Galois field constant multiplication requires far less resources than general multiplications [22].

Based on our evaluation of the AES finalists, the MARS algorithm appeared to be the most resource intensive based on its use of large S-Boxes, and modulo 2^{32} multiplication. As a result, it was conjectured that the MARS algorithm would exhibit lesser performance when compared to the other AES finalists. Due to this evaluation and a lack of development resources, the MARS algorithm was omitted from this study.

4.2 Hardware Architectures

The AES finalists are all comprised of a basic looping structure (some form of either Feistel or substitution-permutation network) whereby data is iteratively passed through a round function. Based on this looping

structure, the following architecture options were investigated so as to yield optimized implementations:

- Iterative Looping
- Loop Unrolling
- Partial Pipelining
- Partial Pipelining with Sub-Pipelining

Iterative looping over a cipher's round structure is an effective method for minimizing the hardware required when implementing an iterative architecture. When only one round is implemented, an n-round cipher must iterate n times to perform an encryption. This approach has a low register-to-register delay but a requires a large number of clock cycles to perform an encryption. This approach also minimizes in general the hardware required for round function implementation but can be costly with respect to the hardware required for round key and S-Box multiplexing. Iterative looping is a subset of loop unrolling in that only one round is unrolled whereas a loop unrolling architecture allows for the unrolling of multiple rounds, up to the total number of rounds required by the cipher. As opposed to an iterative looping architecture, a loop unrolling architecture where all n rounds are unrolled and implemented as a single combinatorial logic block maximizes the hardware required for round function implementation while the hardware required for round key and S-Box multiplexing is completely eliminated. However, while this approach minimizes the number of clock cycles required to perform an encryption, it maximizes the worst case register-to-register delay for the system, resulting in an extremely slow system clock.

A partially pipelined architecture offers the advantage of high throughput rates by increasing the number of blocks of data that are being simultaneously operated upon. This is achieved by replicating the round function hardware and registering the intermediate data between rounds. Moreover, in the case of a full-length pipeline (a specific form of a partial pipeline), the system will output a 128-bit block of ciphertext at each clock cycle once the latency of the pipeline has been met. However, an architecture of this form requires significantly more hardware resources as compared to a loop unrolling architecture. In a partially pipelined architecture, each round is implemented as the pipeline's atomic unit and are separated by the registers that form the actual pipeline. However, many of the AES finalists cannot be implemented using a full-length pipeline due to the large size of their associated round function and S-Boxes, both of which must be replicated n times for an n-round cipher. As such, these algorithms must be implemented as partial pipelines. Additionally, a pipelined architecture can be fully exploited only in modes of operations which do not require feedback of the encrypted data, such as Electronic Code-Book or Counter Mode [3, Section 9.9]. When operating in feedback modes such as Ciphertext Feedback Mode, the ciphertext of one block must be available before the next block can be encrypted. As a result, multiple blocks of plaintext cannot be encrypted in a pipelined fashion when operating in feedback modes. For the remainder of our discussion, feedback mode will be abbreviated as FB and non-feedback mode will be abbreviated as NFB.

Sub-pipelining a (partially) pipelined architecture is advantageous when the round function of the pipelined architecture is complex, resulting in a large delay between pipeline stages. By adding sub-pipeline stages, the atomic function of each pipeline stage is sub-divided into smaller functional blocks. This results in a decrease in the pipeline's delay between stages. However, each sub-division of the atomic function increases the number of clock cycles required to perform an encryption by a factor equal to the number of sub-divisions. At the same time, the number of blocks of data that may be operated upon in NFB mode is increased by a factor equal to the number of sub-divisions. Therefore, for this technique to be effective, the worst case delay between stages will be decreased by a factor of m where m is the number of added sub-divisions. However, if the atomic function of the partially pipelined architecture has a small stage delay, sub-dividing the stage will achieve no significant decrease in the worst case stage delay. In this case, sub-pipelining would result in no significant increase in the system's clock frequency but would increase the logic resources and clock cycles required to perform an encryption, resulting in reduced throughput.

Many FPGAs provide embedded RAM which may be used to replace the round key and S-Box multiplexing hardware. By storing the keys within the RAM blocks, the appropriate key may be addressed based on the current round. However, due to the limited number of RAM blocks, as well as their restricted bit width, this methodology is not feasible for architectures with many pipeline stages or unrolled loops. Those architectures require more RAM blocks than are typically available. Additionally, the switching time for the RAM is more than a factor of three longer than that of a standard CLB slice element, resulting in the RAM element having a lesser speed-up effect on the overall implementation. Therefore, the use of embedded RAM is not considered for this study to maintain consistency between architectural implementations.

5 Architectural Implementation Analysis

For each of the AES finalists, the four architecture options described in Section 4.2 were implemented in VHDL using a bottom-up design and test methodology. The same hardware interface was used for each of the implementations. Round keys are stored in internal registers and all keys must be loaded before encryption may begin. Key loading is disabled until encryption is completed. These implementations yielded a great deal of knowledge in regards to the FPGA suitability of each AES finalist. What follows is a discussion of the knowledge gained regarding each algorithm when implemented using the four architecture types.

5.1 Architectural Implementation Analysis — RC6

When implementing the RC6 algorithm, it was first determined that the RC6 modulo 2^{32} multiplication was the dominant element of the round function in terms of required logic resources. Each RC6 round requires two copies of the modulo 2^{32} multiplier. However, it was found that the RC6 round function does not require a general modulo 2^{32} multiplier. The RC6 multipliers implement the function $A(2A + 1)$ which may be implemented as $2A^2 + A$. Therefore, the multiplication operation was replaced with an array squarer with summed partial products, requiring fewer hardware resources and resulting in a faster implementation. The remaining components of the RC6 round function — fixed and variable shifting, bit-wise XOR, and modulo 2^{32} addition — were found to be simple in structure, resulting in these elements of the round function requiring few hardware resources. While variable shifting operations have the potential to require considerable hardware resources, the 5-bit variable shifting required by the RC6 round function required few hardware resources. Instead of implementing a 32-to-1 multiplexor for each of the thirty-two rotation output bits (controlled by the five shifting bits), a five-level multiplexing approach was used. The variable rotation is broken into five stages, each of which is controlled by one of the five shifting bits. For each rotation output bit of a given stage, a 2-to-1 multiplexor controlled by the stage's shifting bit is used. This implementation requires a total of 160 2-to-1 multiplexors as opposed to the thirty-two 32-to-1 multiplexors required for a one-stage implementation. However, using 2-to-1 multiplexors to form the five-stage barrel-shifter results in an overall implementation that is smaller and faster when compared to the one-stage barrel-shifter implementation as described in [18, Section 3.4]. Finally, it was found that the synthesis tools could not minimize the overall size of a RC6 round sufficiently to allow for a fully unrolled or fully pipelined implementation of the entire twenty rounds of the algorithm within the target FPGA.

As discussed in Section 4.2, implementing a single round of the RC6 algorithm provides the greatest area-optimized solution. Further loop unrolling provided only minor throughput increases as the decrease in the number of cycles per encrypted block was offset by the rapidly decreasing system clock frequency. 2-stage partial pipelining was found to yield the highest throughput when operating in FB mode, outperforming the single round iterative looping implementation by achieving a significantly higher system clock frequency.

When operating in NFB mode, a partially pipelined architecture with two additional sub-pipeline stages was found to offer the advantage of extremely high throughput rates once the latency of the pipeline was met, with the 10-stage partial pipeline implementation displaying the best throughput and results. Based on the delay analysis of the partial pipeline implementations, it was determined that nearly two thirds of

the round function's associated delay was attributed to the modulo 2^{32} multiplier. Therefore, two additional pipeline sub-stages were implemented so as to subdivide the multiplier into smaller blocks, resulting in a total of three pipeline stages per round function. As a result, an increase by a factor of more than 2.5 was seen in the system's clock frequency, resulting in a similar increase in throughput when operating in NFB mode. Further sub-pipelining was not implemented as this would require sub-dividing the adders used to sum the partial products (a non-trivial task) to balance the delay between sub-pipeline stages.

5.2 Architectural Implementation Analysis — Rijndael

When implementing the Rijndael algorithm, it was first determined that the Rijndael S-Boxes were the dominant element of the round function in terms of required logic resources. Each Rijndael round requires sixteen copies of the S-Boxes, each of which is an 8-bit to 8-bit look-up-table, requiring significant hardware resources. However, the remaining components of the Rijndael round function — byte swapping, constant Galois field multiplication, and key addition — were found to be simple in structure, resulting in these elements of the round function requiring few hardware resources. Additionally, it was found that the synthesis tools could not minimize the overall size of a Rijndael round sufficiently to allow for a fully unrolled or fully pipelined implementation of the entire ten rounds of the algorithm within the target FPGA.

Surprisingly, a one round partially pipelined implementation with one sub-pipeline stage provided the most area-optimized solution. As compared to a one-stage implementation with no sub-pipelining, the addition of a sub-pipeline stage afforded the synthesis tool greater flexibility in its optimizations, resulting in a more area efficient implementation. While 2-stage loop unrolling was found to yield the highest throughput when operating in FB mode, the measured throughput was within 10% of the single stage implementation. Due to the probabilistic nature of the place-and-route algorithms, one can expect a variance in performance based on differences in the starting point of the process. When performing this process multiple times, known as multi-pass place-and-route, it is likely that the single round implementation would achieve a throughput similar to that of the 2-stage loop unrolled implementation.

When operating in NFB mode, partial pipelining was found to offer the advantage of extremely high throughput rates once the pipeline latency was met, with the 5-stage partial pipeline implementation displaying the best throughput results. While Rijndael cannot be implemented using a fully pipelined architecture due to the large size of the round function, significant throughput increases were seen as compared to the loop unrolling architecture.

Sub-pipelining of the partially pipelined architectures was implemented by inserting a pipeline sub-stage within the Rijndael round function. Based on the delay analysis of the partial pipeline implementations, it was determined that nearly half of the round function's associated delay was attributed to the S-Box substitutions. Therefore, the additional pipeline sub-stage was implemented so as to separate the S-Boxes from the rest of the round function. As a result, an increase by a factor of nearly 2 was seen in the system's clock frequency, resulting in a similar increase in throughput when operating in NFB mode. Further sub-pipelining was not implemented as this would require sub-dividing the S-Boxes (a non-trivial task) to balance the delay between sub-pipeline stages.

5.3 Architectural Implementation Analysis — Serpent

When implementing the Serpent algorithm, it was first determined that since the Serpent S-Boxes are relatively small (4-bit to 4-bit), it is possible to implement them using combinational logic as opposed to memory elements. Additionally, the S-Boxes map extremely well to the Xilinx CLB slice, which is comprised of 4-bit look-up-tables, allowing one S-Box to be implemented in a total of two CLB slices, yielding a compact implementation which minimizes routing between CLB slices. Finally, the components of the Serpent round function — key masking, S-Box substitution, and linear transformation — were found to be simple in structure, resulting in the round function requiring few hardware resources.

Implementing a single round of the Serpent algorithm provides the greatest area-optimized solution. However, a significant performance improvement was achieved by performing 8-round loop unrolling, removing the need for S-Box multiplexing hardware as one copy of each possible S-Box grouping is now included within one of the eight rounds. This amount of loop unrolling achieved a significant performance increase with little increase in hardware resources due to the compact nature of the Serpent round function. As expected, unrolling thirty-two rounds of the Serpent algorithm resulted in a lesser performance when compared to the eight round implementation. Implementing the thirty-two rounds of the algorithm in combinatorial logic severely hampered the overall clock frequency of the system, overriding the performance increase caused by the removal of the multiplexing hardware required to switch between keys.

When operating in NFB mode, a full-length pipelined architecture was found to offer the advantage of extremely high throughput rates once the latency of the pipeline was met, outperforming smaller partially pipelined implementations. In the fully pipelined architecture, all of the elements of a given round function are implemented as combinatorial logic. Other AES finalists cannot be implemented using a fully pipelined architecture due to the larger round functions. However, due to the small size of the Serpent S-Boxes (4-bit look-up-tables), the cost of S-Box replication is minimal in terms of the required hardware.

Finally, sub-pipelining of the partially pipelined architectures was determined to yield no throughput increase. Because the round function components are all simple in structure, there is little performance to be gained by subdividing them with registers in an attempt to reduce the delay between stages. As a result, the increase in the system's clock frequency would not outweigh the increase in the number of clock cycles required to perform an encryption, resulting in a performance degradation.

5.4 Architectural Implementation Analysis — Twofish

When implementing the Twofish algorithm, it was first determined that the synthesis tools were unable to minimize the Twofish S-Boxes to the extent of other AES finalist algorithms due to the S-Boxes being key-dependent. Therefore, the overall size of a Twofish round was too large to allow for a fully unrolled or fully pipelined implementation of the algorithm within the target FPGA. Moreover, the key-dependent S-Boxes were found to require nearly half of the delay associated with the Twofish round function.

As expected, implementing a single round of the Twofish algorithm provides the greatest area-optimized solution in terms of total CLB slices required for the implementation. Additional loop unrolling provided minor throughput increases as the decrease in the number of cycles per encrypted block was offset by the rapidly decreasing system clock frequency. However, single stage partial pipelining with one sub-pipeline stage was found to yield the best throughput and when operating in feedback mode. With a small increases in the required hardware resources, the sub-pipelined architecture was able to reach a significantly faster system clock frequency as compared to the loop unrolling and partial pipeline implementations.

When operating in NFB mode, a partially pipelined architecture was found to offer the advantage of extremely high throughput rates once the latency of the pipeline was met, with the 8-stage partial pipeline implementation displaying the best throughput results. While Twofish cannot be implemented using a fully pipelined architecture due to the large size of the round function, significant throughput increases were seen as compared to the loop unrolling architecture.

Finally, sub-pipelining of the partially pipelined architectures was implemented by inserting a pipeline sub-stage within the Twofish round function. Based on the delay analysis of the partial pipeline implementations, it was determined that nearly half of the round function's associated delay was attributed to the S-Box substitutions. Therefore, the additional pipeline sub-stage was implemented so as to separate the S-Boxes from the rest of the round function. As a result, an increase by a factor of nearly 2 was seen in the system's clock frequency, resulting in a similar increase in throughput when operating in NFB mode. Further sub-pipelining was not implemented as this would require sub-dividing the S-Boxes (a non-trivial task) to balance the delay between sub-pipeline stages.

6 Performance Evaluation

Tables 2 and 3 detail the throughput measurements for the implementations of the three architecture types for each of the AES finalists for both NFB and FB mode. The architecture types — loop unrolling (LU), full or partial pipelining (PP), and partial pipelining with sub-pipelining (SP) — are listed along with the number of stages and (if necessary) sub-pipeline stages in the associated implementation; e.g., LU-4 implies a loop unrolling architecture with four rounds, while SP-2-1 implies a partially pipelined architecture with two stages and one sub-pipeline stage per pipeline stage. As a result, the SP-2-1 architecture implements two rounds of the given cipher with a total of two stages per round. Throughput is calculated as:

$$Throughput \quad := \quad (128 \text{ Bits } * \text{ Clock Frequency})/(\text{Cycles Per Encrypted Block})$$

Note that the implementation of a one stage partial pipeline architecture, an iterative looping architecture, and a one round loop unrolled architecture are all equivalent and are therefore not listed separately. Also note that the computed throughput for implementations that employ any form of hardware pipelining (as discussed in Section 4) are made assuming that the pipeline latency has been met.

The number of CLBs required as well as the maximum operating frequency for each implementation was obtained from the Xilinx report files. Note that the Xilinx tools assume the absolute worst possible operating conditions — highest possible operating temperature, lowest possible supply voltage, and worst-case fabrication tolerance for the speed grade of the FPGA [23]. As a result, it is common for actual implementations to achieve slightly better performance results than those specified in the Xilinx report files.

While this study focuses on high throughput implementations, the hardware resources required to achieve this throughput is also a critical parameter. No established metric exists to measure the hardware resource costs associated with the measured throughput of an FPGA implementation. Two area measurements of FPGA utilization are readily apparent — logic gates and CLB slices. It is important to note that the logic gate count does not yield a true measure of actual FPGA utilization. Hardware resources within CLB slices may not be fully utilized by the place-and-route software so as to relieve routing congestion. This results in an increase in the number of CLB slices without a corresponding increase in logic gates. To achieve a more accurate measure of chip utilization, CLB slice count was chosen as the most reliable area measurement. Therefore, to measure the hardware resource cost associated with an implementation's resultant throughput, the Throughput Per Slice (TPS) metric is used. We defined TPS as:

$$TPS \quad := \quad (\text{Encryption Rate})/(\#\text{ CLB Slices Used})$$

Therefore, the optimal implementation will display the highest throughput and have the largest TPS. Note that the TPS metric behaves inversely to the classical time-area (TA) product.

When comparing implementations using the TPS and throughput metrics, it is required that the architectures are implemented on the same FPGA. Different FPGAs within the same family yield different timing results as a function of available logic and routing resources, both of which change based on the die size of the FPGA. Additionally, it is impossible to legitimately compare FPGAs from separate families as each family of FPGAs has a unique architecture which greatly affects the measured throughput and TPS. Finally, it is critical to note that throughput (and therefore TPS) may not scale linearly based on the number of rounds implemented for the three architecture types detailed in Section 4.1. As a result, it is imperative that multiple implementations be examined for each architecture type, varying the round count to determine the optimal number of rounds per implementation.

Algorithm	Architecture	Slices	Clock Frequency (MHz)	Cycles per Block	Throughput (Mbit/s)
RC6	LU-1	2638	13.8	20	88.5
RC6	LU-2	3069	7.3	10	94.0
RC6	LU-4	4070	3.7	5	94.8
RC6	LU-5	4476	2.9	4	92.2
RC6	LU-10	6406	1.5	2	97.4
RC6	PP-2	3189	19.8	10	253.0
RC6	PP-4	4411	12.3	5	315.5
RC6	PP-5	4848	12.1	4	386.7
RC6	PP-10	7412	13.3	2	848.1
RC6	SP-1-1	2967	26.2	20	167.6
RC6	SP-2-1	3709	26.4	10	337.8
RC6	SP-4-1	5229	24.6	5	629.8
RC6	SP-5-1	5842	25.8	4	825.2
RC6	SP-10-1	8999	26.6	2	1704.6
RC6	SP-1-2	3134	39.1	20	250.0
RC6	SP-2-2	4062	38.9	10	497.4
RC6	SP-4-2	5908	31.3	5	802.3
RC6	SP-5-2	6415	33.3	4	1067.0
RC6	**SP-10-2**	**10856**	**37.5**	**2**	**2397.9**
Rijndael	LU-1	3528	25.3	11	294.2
Rijndael	LU-2	5302	14.1	6	300.1
Rijndael	LU-5	10286	5.6	3	237.4
Rijndael	PP-2	5281	23.5	5.5	545.9
Rijndael	PP-5	10533	20.0	2.2	1165.8
Rijndael	SP-1-1	3061	40.4	10.5	491.9
Rijndael	SP-2-1	4871	38.9	5.25	949.1
Rijndael	**SP-5-1**	**10992**	**31.8**	**2.1**	**1937.9**
Serpent	LU-1	5511	15.5	32	61.9
Serpent	LU-8	7964	13.9	4	444.2
Serpent	LU-32	8103	2.4	1	312.3
Serpent	PP-8	6849	30.4	4	971.8
Serpent	**PP-32**	**9004**	**38.0**	**1**	**4860.2**
Twofish	LU-1	2666	13.0	16	104.2
Twofish	LU-2	3392	7.1	8	113.6
Twofish	LU-4	4665	3.3	4	106.8
Twofish	LU-8	6990	1.7	2	108.1
Twofish	PP-2	3519	11.9	8	190.4
Twofish	PP-4	5044	11.5	4	369.3
Twofish	PP-8	7817	10.8	2	689.5
Twofish	SP-1-1	3053	29.9	16	239.2
Twofish	SP-2-1	3869	28.6	8	457.1
Twofish	SP-4-1	5870	27.3	4	872.3
Twofish	**SP-8-1**	**9345**	**24.8**	**2**	**1585.3**

Table 2: AES finalist performance evaluation — non-feedback mode

Algorithm	Architecture	Slices	Clock Frequency (MHz)	Cycles per Block	Throughput (Mbit/s)
RC6	LU-1	2638	13.8	20	88.5
RC6	LU-2	3069	7.3	10	94.0
RC6	LU-4	4070	3.7	5	94.8
RC6	LU-5	4476	2.9	4	92.2
RC6	LU-10	6406	1.5	2	97.4
RC6	**PP-2**	**3189**	**19.8**	**20**	**126.5**
RC6	PP-4	4411	12.3	20	78.9
RC6	PP-5	4848	12.1	20	77.3
RC6	PP-10	7412	13.3	20	84.8
RC6	SP-1-1	2967	26.2	40	83.8
RC6	SP-2-1	3709	26.4	40	84.5
RC6	SP-4-1	5229	24.6	40	78.7
RC6	SP-5-1	5842	25.8	40	82.5
RC6	SP-10-1	8999	26.6	40	85.2
RC6	SP-1-2	3134	39.1	60	83.3
RC6	SP-2-2	4062	38.9	60	82.9
RC6	SP-4-2	5908	31.3	60	66.9
RC6	SP-5-2	6415	33.3	60	71.1
RC6	SP-10-2	10856	37.5	60	79.9
Rijndael	LU-1	3528	25.3	11	294.2
Rijndael	**LU-2**	**5302**	**14.1**	**6**	**300.1**
Rijndael	LU-5	10286	5.6	3	237.4
Rijndael	PP-2	5281	23.5	11	273.0
Rijndael	PP-5	10533	20.0	11	233.2
Rijndael	SP-1-1	3061	40.4	21	246.0
Rijndael	SP-2-1	4871	38.9	21	237.3
Rijndael	SP-5-1	10992	31.8	21	193.8
Serpent	LU-1	5511	15.5	32	61.9
Serpent	**LU-8**	**7964**	**13.9**	**4**	**444.2**
Serpent	LU-32	8103	2.4	1	312.3
Serpent	PP-8	6849	30.4	32	121.5
Serpent	PP-32	9004	38.0	32	151.9
Twofish	LU-1	2666	13.0	16	104.2
Twofish	LU-2	3392	7.1	8	113.6
Twofish	LU-4	4665	3.3	4	106.8
Twofish	LU-8	6990	1.7	2	108.1
Twofish	PP-2	3519	11.9	16	95.2
Twofish	PP-4	5044	11.5	16	92.3
Twofish	PP-8	7817	10.8	16	86.2
Twofish	**SP-1-1**	**3053**	**29.9**	**32**	**119.6**
Twofish	SP-2-1	3869	28.6	32	114.3
Twofish	SP-4-1	5870	27.3	32	109.0
Twofish	SP-8-1	9345	24.8	32	99.1

Table 3: AES finalist performance evaluation — feedback mode

Alg.	Arch.	Throughput (Gbit/s)		Slices	TPS
RC6	SP-10-2	**2.40**		10856	220881
Rijndael	SP-5-1	**1.94**		10992	176297
Serpent	PP-32	**4.86**		9004	539778
Twofish	SP-8-1	**1.59**		9345	169639

Table 4: AES finalist performance evaluation — non-feedback mode speed-optimized implementations

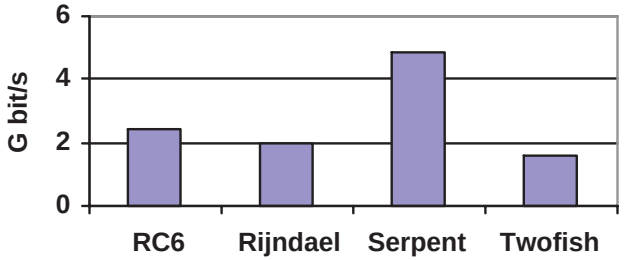


Figure 1: Best throughput — non-feedback mode

Alg.	Arch.	Throughput (Mbit/s)		Slices	TPS
RC6	PP-2	**126.5**		3189	39662
Rijndael	LU-2	**300.1**		5302	56605
Serpent	LU-8	**444.2**		7964	55771
Twofish	SP-1-1	**119.6**		3053	39169

Table 5: AES finalist performance evaluation — feedback mode speed-optimized implementations

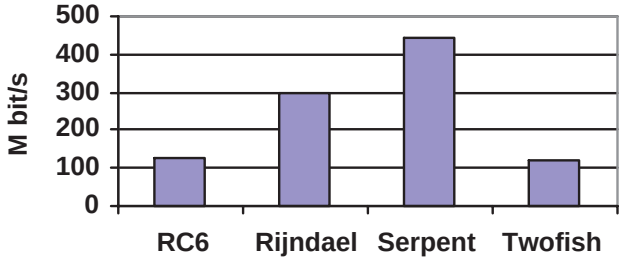


Figure 2: Best throughput — feedback mode

Tables 4 and 5 detail the optimal implementations of the AES finalists in both FB and NFB modes. Additionally, TPS is also shown for each of the implementations. It is critical to note that for the purposes of this study, the optimal implementation for an AES finalist is defined to yield the highest throughput. *As previously discussed, the synthesis tools were set to optimize for speed to guarantee that the highest throughputs would be achieved for each implementation. However, should an optimal implementation be defined based on either TPS or area, the implementation results shown in Tables 2 and 3 (and, as a result, those shown in tables 4 and 5 as well) are no longer representative of the best possible implementations for the architectures studied. To achieve a true representation that defines optimality based on either TPS or area, synthesis must be performed with the tools set to optimize for area.* While an area-efficiency analysis of the AES finalists warrants investigation, it is beyond the scope of this study.

Based on the data shown in Tables 4 and 5, the Serpent algorithm clearly outperforms the other AES finalists in both modes of operation. As compared to its nearest competitor, Serpent exhibits a throughput increase of a factor 2.2 in NFB mode and a factor 1.5 in FB mode. Interestingly, RC6, Rijndael, and Twofish

all exhibit similar performance results in NFB mode. However, Rijndael exhibits significantly improved performance in FB mode as compared to RC6 and Twofish, although it is still 50% slower than Serpent.

One of the main findings of our investigation, namely that Serpent appears to be especially well suited for an FPGA implementation from a performance perspective, seems especially interesting considering that Serpent is clearly not the fastest algorithm with respect to most software comparisons [5]. Another major result of our study is that all four algorithms considered easily achieve Gigabit encryption rates with standard commercially available FPGAs. The algorithms are at least one order of magnitude faster than the best reported software realizations. These speed-ups are essentially achieved by parallelization (pipelining and sub-pipelining) of the loop structure and by wide operand processing (e.g., processing of 128 bits in once clock cycle), both of which are not feasible on current processors. We would like to stress that the pipelined architectures cannot be used to their maximum ability for modes of operation which require feedback (CFB, OFB, etc.) However we believe that for many applications which require high encryption rates, non-feedback modes (or modified feedback modes such as interleaved CFB [3, Section 9.12]) will be the modes of choice. Note that the Counter Mode grew out of the need for high speed encryption of ATM networks which required parallelization of the encryption algorithm.

7 Conclusions

The importance of the Advanced Encryption Standard and the significance of high throughput implementations of the AES finalists has been examined. A design methodology was established which in turn led to the architectural requirements for a target FPGA. The core operations of the AES finalists were identified and multiple architecture options were discussed. The implementation of each architecture option for each of the AES finalists was analyzed to determine their suitability for hardware implementation. Based on the implementation results, the best speed-optimized implementations were identified for each AES finalist in both non-feedback and feedback modes. Upon comparison, it was determined that the Serpent algorithm yielded the best performance in both modes, where best performance was defined strictly as the highest throughput. The Serpent algorithm outperforms its nearest competitor by a factor of 2.2 in non-feedback mode and by a factor of 1.5 in feedback mode.

8 Acknowledgement

We would like to thank Pawel Chodowiec and Kris Gaj from George Mason University for their helpful discussion and the VHDL code modules that were provided to assist in the implementation of some of the AES finalists. We would also like to thank Alan Martello from the University of Pittsburgh for his public-domain VHDL code module that was used in implementation of the AES finalists.

References

[1] D. Stinson, *Cryptography, Theory and Practice*. Boca Raton, FL: CRC Press, 1995.

[2] National Institute of Standards and Technology (NIST), *Second Advanced Encryption Standard (AES) Conference*, (Rome, Italy), March 1999.

[3] B. Schneier, *Applied Cryptography*. John Wiley & Sons Inc., 2nd ed., 1995.

[4] R. Doud, "Hardware Crypto Solutions Boost VPN," *EETimes*, pp. 57–64, April 1999.

[5] B. Gladman, "Implementation Experience with AES Candidate Algorithms," in *Proceedings: Second AES Candidate Conference (AES2)*, (Rome, Italy), March 1999.

[6] J. Kaps and C. Paar, "Fast DES Implementations for FPGAs and its Application to a Universal Key-Search Machine," in *5th Annual Workshop on Selected Areas in Cryptography (SAC '98)* (S. Tavares and H. Meijer, eds.), vol. LNCS 1556, (Queen's University, Kingston, Ontario, Canada), Springer-Verlag, August 1998.

[7] E. Biham, "A Fast New DES Implementation in Software," in *Fast Software Encryption. 4th International Workshop, FSE'97 Proceedings*, (Berlin), pp. 260–272, Springer-Verlag, 1997. Lecture Notes in Computer Science Volume 1267.

[8] A. Pfitzmann and R. Assman, "More Efficient Software Implementations of (Generalized) DES," *Computers & Security*, vol. 12, no. 5, pp. 477–500, 1993.

[9] J. Hughes, "Implementation of NBS/DES Encryption Algorithm in Software," in *Colloquium on Techniques and Implications of Digital Privacy and Authentication Systems*, 1981.

[10] D. Runje and M. Kovac, "Universal Strong Encryption FPGA Core Implementation," in *Proceedings of Design, Automation, and Test in Europe*, (Paris, France), pp. 923–924, February 1998.

[11] O. Mencer, M. Morf, and M. Flynn, "Hardware Software Tri-Design of Encryption for Mobile Communication Units," in *Proceedings of International Conference on Acoustics, Speech, and Signal Processing*, (Seattle, WA), May 1998.

[12] A. Elbirt, "An FPGA Implementation and Performance Evaluation of the CAST-256 Block Cipher," Technical Report, Cryptography and Information Security Group, Electrical and Computer Engineering Department, Worcester Polytechnic Institute, Worcester, MA, May 1999.

[13] M. Riaz and H. Heys, "The FPGA Implementation of RC6 and CAST-256 Encryption Algorithms," in *accepted for CCECE'99*, (Edmonton, Alberta, Canada), 1999.

[14] C. Phillips and K. Hodor, "Breaking the 10k FPGA Barrier Calls For an ASIC-Like Design Style," *Integrated System Design*, 1996.

[15] R. Anderson, E. Biham, and L. Knudsen, "Serpent: A Proposal for the Advanced Encryption Standard," in *First Advanced Encryption Standard (AES) Conference*, (Ventura, CA), 1998.

[16] B. Schneier, J. Kelsey, D. Whiting, D. Wagner, and C. Hall, "Twofish: A 128-Bit Block Cipher," in *First Advanced Encryption Standard (AES) Conference*, (Ventura, CA), 1998.

[17] J. Daemen and V. Rijmen, "AES Proposal: Rijndael," in *First Advanced Encryption Standard (AES) Conference*, (Ventura, CA), 1998.

[18] R. Rivest, M. Robshaw, R. Sidney, and Y. Yin, "The RC6^{TM} Block Cipher," in *First Advanced Encryption Standard (AES) Conference*, (Ventura, CA), 1998.

[19] Xilinx Inc., *Virtex 2.5V Field Programmable Gate Arrays*, 1998.

[20] B. Chetwynd, "Universal Block Cipher Module: Towards a Generalized Architectures for Block Ciphers," Master's thesis, Worcester Polytechnic Institute, Worcester, MA, November 1999.

[21] S. Brown and J. Rose, "FPGA and CPLD Architectures: A Tutorial," in *IEEE Design & Test of Computers*, vol. 13, no. 2, pp. 42–57, 1996.

[22] C. Paar, "Optimized Arithmetic for Reed-Solomon Encoders," in *1997 IEEE International Symposium on Information Theory*, (Ulm, Germany), p. 250, June 29 – July 4 1997.

[23] P. Alfke, "Xilinx M1 Timing Parameters." electronic mail personal correspondance, December 1999.

A Comparison of the AES Candidates Amenability to FPGA Implementation

Nicholas Weaver, John Wawrzynek
{nweaver,johnw}@cs.berkeley.edu*

March 15, 2000

Abstract

The 5 final AES candidates, MARS, RC6, Rijndael, Serpent, and Twofish, are all intended to run well both on hardware and software implementations. However, the different algorithms may result in significant differences in cost and performance when implemented on FPGAs or in small custom devices. This document discusses the various algorithms from the perspective of a potential FPGA implementer. Rijndael and Twofish are excellent candidates from a hardware designer's viewpoint, while MARS is particularly expensive and inefficient.

1 Introduction

The 5 final AES candidates, MARS[2], RC6[6], Rijndael[4], Serpent[1], and Twofish[7], are all designed to run efficiently on a wide variety of hardware and software. However, the candidates vary in their amenability to hardware implementations. In this paper we estimate the relative cost and performance for various possible implementations of the different AES algorithms. Although no actual implementations are realized, it is straightforward to estimate the cost of various possible realizations of the AES candidates.

2 Observations on the candidates

The following observations will be justified throughout the paper.

MARS is not a very suitable cipher for a hardware implementation. The three separate round types, the use of both an expensive multiplier and numerous large S-box references, and a complicated subkey generation, all combine to make it a poor candidate.

RC6, though it uses comparatively expensive operations, is a reasonable candidate unless subkey generation is important. The ability to reasonably reduce the hardware requirements without sacrificing too much performance is present, a useful feature when a low cost implementation is desired. However, the subkey generation, which has a tight dependency and needs to visit elements multiple times, poses a considerable challenge for any application which needs to change keys frequency.

Rijndael is probably the best candidate when subkey flexibility isn't essential. All operations are highly parallel but comparatively inexpensive in hardware, and the subkey generation is both fast and compact. However, the additional cost of creating a separate datapath if decryption is required somewhat hampers the design, and the subkey generation may still have an impact, depending on the application.

Serpent, surprisingly enough, is not the best candidate from a hardware standpoint. Although it uses very short operations which map naturally to hardware, 32 instances of

*This work was suppored by DARPA, contract number DABT63-96-C-0048. Further support comes from the California State MICRO Program.

each of the 8 types of S-boxes quickly add up, and if a compact implementation is desired, the bandwidth is considerably reduced. Also, there is essentially no sharing between encryption and decryption pipelines.

Twofish is the best overall from a hardware viewpoint. Although not as fast as Rijndael and Serpent, the ability to perform encryption and decryption with a trivially modified pipeline is quite valuable. Also, there is a nice tradeoff space between area and performance. If subkeys are not changed, the subkey generation can largely be folded into the pipeline. If area is still tight, the pipeline can be folded in half. However, if subkeys are changed often and performance is critical, the ability to change subkeys from block to block with almost no performance penalty, whether encrypting or decrypting, is of significant potential benefit. This degree of flexibility is unique to Twofish, and is a very desirable property.

3 Possible implementation techniques

There are 3 primary criteria when measuring the candidates using a hardware metric: latency, bandwidth, and area. Latency is the amount of time required to encrypt a single block of data. If the cipher is operating in CFB or similar modes, the latency of encryption may be the critical factor. Bandwidth is the number of blocks which can be computed in a given period of time. If there is no feedback on the ciphertext, such as in ECB mode, bandwidth indicates how fast data can be encrypted. Area is a specific metric, which generally suggests the cost for an implementation. Lower area is generally beneficial, as this allows lower cost parts to be used.

The implementation fabric being considered is the Xilinx Virtex[9] Field Programmable Gate Array, which consists of an array of 4 input lookup tables (4-LUTs)[1] and associated flip flops, plus a perimeter of medium sized, dual ported, 512 byte BlockRAM memories[2]. Each 4-input lookup table can also act as a 16-bit RAM, for storing temporary values.

Embedded, small to medium sized memory blocks are becoming ubiquitous on modern FPGAs, although many older devices (such as the Xilinx 4000 series) lack such features. Thus, the use of such memories needs to be considered separately. It is, however, safe to assume that practically all future devices will have such capabilities.

It is comparatively easy to estimate the size of a hand layed out datapath for these applications, as the dataflows are suitably regular to allow the functional units to be packed together. The cost of the control logic is not considered, because for the AES candidates the primary cost is the datapath.

Similarly, the cost of generating the encryption subkeys is considered separately; for some algorithms subkey generation may be better implemented on a small microcontroller[3]. Some applications may use constant subkeys or subkeys which change only rarely, in which case subkey generation time is not a concern. However, in other applications where encryption keys may change on a packet-by-packet basis, subkey generation can become the dominant factor in the time it takes to encrypt a block.

Although the sketches described are geared towards FPGAs, a good rule of thumb is that, except for memories, logic in an FPGA takes roughly ten times the silicon area of an ASIC, while using very similar design techniques. Thus, these implementation techniques and relative cost metrics could carry over into the ASIC realm.

There are three common hardware implementation techniques considered. These are small microcoded datapaths; a pipelined, single or multiple round, C-slow[4] structure; and a fully unrolled datapath.

A microcoded datapath may be the most compact design, but often suffers from very poor bandwidth. It consists of a register file, a datapath of several functional units cus-

[1] A 4-input lookup table can realize any boolean of 4 inputs

[2] These are small, 8 address, 16b wide memories, which have two separate address and data ports. This allows two separate memory locations to be written or read in a single cycle.

[3] There is a current trend towards FPGAs with microcontrollers, such as the Triscent parts[8].

[4] Two paragraphs further defines C-slow. Be patient.

tomized to the application at hand, and a small program (usually contained in a small ROM) which controls the datapath. The problem with such implementations is that the aggregate bandwidth is usually very low and the design is unable to utilize the parallelism inherent in the algorithm.

A C-slow datapath implements a single round or group of rounds, separated into C pipeline stages which operate on different blocks. This allows for considerably higher bandwidth then a single iterative round, as C independent blocks can be processed through the pipeline. The number C is usually chosen to match the desired clock rate. A C-slow pipeline can run at a high clock rate and adding more register stages can allow an even higher clock rate (and therefore higher bandwidth) without affecting the latency[5]. Furthermore, since more operations can be done in parallel, this technique may improve the overall latency when compared with a microcoded implementation.

For most algorithms, a C-slow, single round pipeline should require roughly the same area as a microcoded datapath or an unpipelined round, while offering a considerable improvement in bandwidth, as the functional units are more highly utilized. Thus, a C-slow technique should always be utilized unless such a design simply can not be implemented in the available area or the implementation fabric is flip-flop poor.

A fully unrolled datapath, where each round is separatly implemented in hardware, can be similarly pipelined to run at a high speed. This offers essentially no latency advantage over a C-slow datapath, but allows for the maximum bandwidth possible. The number of pipeline stages is chosen in a similar way to the C-slow implementations, to provide operation which matches a target clock rate. The area cost and available bandwidth of a full pipeline are a simple multiple of the area and bandwidth for a C-slow implementation, so unrolled pipelines are not considered in detail in this analysis.

In general, we will attempt to roughly estimate the number of pipeline stages which would be required to allow a Virtex implementation to run at a 50 MHz clock cycle. A typical, modern, midsized FPGA such as the Virtex XCV200 contains 5000 4-LUTs and 14 BlockRAMs, while a typical compact, low cost FPGA such as the Xilinx Spartan2[6] XC2S50 contains 1,700 LUTs and 8 BlockRAMs.

Though these implementation sketches are for a particular FPGA fabric, these comparisons should carry over[7] to other FPGAs and small ASICs.

4 Cryptographic core

The cost for the different implementation's cryptographic cores were estimated by summarizing the costs of their respective subcomponents.

Serpent is the best from a pure performance viewpoint in hardware, although Rijndael and Twofish are close to it in performance and area/performance. The significant problem with Serpent is there is a significant minimum size for the implementation to be effective. MARS is comparatively awkward, requiring both a relatively large amount of logic and a large amount of ROM for table lookups.

The other problem with Rijndael and Serpent is that separate pipelines are required for encryption and decryption. Having to implement separate pipelines for encryption and decryption doubles the area of an implementation if both operations are required.

A RC6 pipeline can be easily modified to perform both encryption and decryption by replacing the adders with adder/subtracters (a no cost or very low-cost transformation). The Feistel basis of MARS and Twofish allow a slightly tweaked pipeline to handle both rolls effectively.

[5] This technique has a limit of the setup and hold time of the flip flop, and the granularity at which different paths may require different latencies.

[6] A low cost revision of the Virtex

[7] Although with some caveats, usually dealing with local memories and the use of tristate buffers to implement wide muxes, which may not be present in other FPGA fabrics

Algorithm	Implementation	Latency (cycles)	Bandwidth (blocks/cycle)	Size (4-LUTs)	Size (BlockRAM)
MARS	Microcoded datapath	480	1/480	770	8
	6-Slow, single round	190	1/16	1500	12
RC6	5-Slow, single round	102	1/20	1700	0
	8-Slow, folded round	164	1/40	950	0
Rijndael	2-Slow, single round	20	1/10	780	8
Serpent	8 slow, 8 round	32	1/4	3800	0
	single round	32	1/32	1600	0
Twofish	3-Slow, single round	50	1/16	1350	0
	4-Slow, folded round	66	1/32	870	0

Figure 1: A comparison of the implementation costs for the various algorithms

A mixed Rijndael pipeline can share the S-boxes by separating the transformation from the S-box, which adds a small step and some area. However, this approach still requires a completely different column mixing step and therefore a fairly significant area cost to handle both encryption and decryption. Some implementations would probably just use separate pipelines, since depending on the implementation technology, the cost of the S-boxes may be dwarfed by the remaining costs.

Serpent can share almost no area between encryption and decryption, since it is dependent on inverse-sboxes and inverse-transformations for decryption. This essentially doubles the cost of a Serpent device which performs both encryption and decryption.

4.1 MARS

MARS is unfortunately comparatively costly to implement on small devices, as a microcoded datapath is more compact then a C-slow pipeline. There are also several comparatively expensive elements: variable rotations, the numerous, large S-box references, and the 32 bit multiplier. Since the multiplier and rotates are on the critical path and can not have their latencies hidden, a fast array multiplier and a barrel rotator are necessary to achieve good performance.

4.1.1 Microcoded datapath

A microcoded datapath would require 4 BlockRAMs for a 32 bit, 2 read, one write port register file for a scratchpad and subkey storage, another 4 BlockRAMs used as ROMs to store the S-Box, 32 LUTs for the XOR, 32 LUTs for the adder/subtracter, 160 LUTs for a barrel rotator, and 512 LUTs for an array multiplier[8]. Thus, this datapath requires roughly 8 BlockRAMs and 768 LUTs.

Assuming a single cycle latency for all operations but the multiplier, and assuming 3 cycles for the multiplier, it would take roughly 13 operations for each round of forward mixing, 18 for one round of the cryptographic core, and 12 rounds for the backwards mixing. Thus, it would require at least 480 cycles of latency for a single encryption. Furthermore, it is very difficult to run more than one or two blocks through such a datapath.

4.2 Full Round, C-slow

A C-slow pipeline is less compact on MARS when compared to other algorithms, due to the expense of various components and the 3 separate round types. The forward mixing would require 4 BlockRAMs for the S-box halves and 172 LUTs for the logic. The core would

[8] A booth encoded, shift and add multiplier could probably be constructed for only 64 to 128 LUTs, but would require 16 cycles/multiply instead of 3 cycles cycles

require roughly 4 BlockRAMs for the sbox, 64 LUTs to store the subkeys, 512 LUTs to compute R, 172 LUTs to compute M, and 172 LUTs to compute L, plus an additional 150 LUTs to compute B, C, and D, for a total of 1070 LUTs and 4 BlockRAMs for the core. The final mixing would require another 4 BlockRAMs and 172 LUTs for logic, for a total of 12 BlockRAMs and nearly 1500 LUTs. The number of 4-LUTs is reasonable, but the large number of S-box references are a considerable expense, making this implementation prohibitive on devices without local memories to use for the S-boxes.

In order to run the central core at a desired 50 MHz, it would probably be necessary to run it 6-slow[9], with the forward and backward mixings running 3 slow This would require 192 cycles of latency to encrypt a single block, but would produce one block every 16 clock cycles.

The biggest problem with MARS is the numerous references to the large S-Boxes. If a bandwidth-oriented implementation is desired, the number of S-Box references becomes very expensive. The 32 bit, modulo 2^{32} multiplier is expensive, but not prohibitively slow. Finally, the 2 variable rotations are moderately expensive operations. The biggest expense is the three different round types: although not a concern for a software implementor, it is a significant handicap for hardware designs.

4.3 RC6

RC6 uses operations which, while inexpensive in a modern microprocessor, are moderately expensive in hardware. A 32 bit, modulo 2^{32} multiplier require 512 LUTs, and a 32 bit rotator would require 160 LUTs to accomplish. However, there is a nice ability to trade off performance for area in this design.

4.3.1 Full round, C-slow

The most straightforward, compact implementation of RC6 is a single round, C-slow implementation. The initial and final keys are best stored in registers, while the remaining keys would fit in 128 LUTs. The MUXes on the start of the pipeline (to select between the input and the result from the previous round) require 128 LUTs, and the input and output whitening each require 64 LUTs.

The pipeline for the round itself would need 512 LUTs for each F function to perform the 32 bit multiplication. The variable rotations require 160 LUTs but can be combined with the XOR operation, and each of the subkey additions requires 32 LUTs. When added to the hardware required for muxing plus the initial and final adders, the total comes to roughly 1700 LUTs for the pipeline.

It should take 3 cycles to perform the F function, another cycle for the rotation, and a final cycle for the subkey addition, suggesting that a 5-slow pipeline would be sufficient. This would require 102 cycles latency to produce a result but would be able to produce a result every 20 cycles.

4.3.2 Compact, half-round, C-slow design

There are some tricks which can be used for a more compact RC6 design. Since both sides of a round are identical, the implementer could build a half-round, C-slow implementation which folds the two halves together. This roughly cuts the resource requirements and bandwidth in half, and adds three cycles of latency per round in order to exchange t and u and to perform the exchange at the end of each round, with an addititonal cycle of padding to implement a round in an even number of clock periods. In a case where bandwidth is as important as latency while resources are heavily constrained, this technique would be significantly prefered over a microcoded datapath.

[9] 3 cycles to compute R, 1 cycle to compute M, 1 to compute L, and 1 cycle to compute the new values of B, C, and D

The additional costs of such a datapath are one extra cycle for each swap and one cycle for padding, making the pipeline 8-slow and uping the latency to 164 cycles, and the bandwidth reduced to one block every 40 cycles. This allows the core to be almost cut in half, to 870 LUTs, with another 32 LUTs to store the remaining subkeys. Also, an additional 40 LUTs are required for various MUXes, and the subkey storage and whitening remain unchanged. Thus, the cost of such an implementation would be roughly 950 LUTs.

4.4 Rijndael

Rijndael's number of rounds depends on the key size. For this analysis both the block and key size are 128 bits. Rijndael has a high degree of parallelism, with very short operations and a small number of rounds, which makes it one of the fastest candidates for a hardware implementer.

The Mix-column operation of Rijndael would require 8 LUTs for the accumulation of each byte, with each multiplication probably reducable into 8 LUTs similar to the technique in [3]. Thus, the entire mix column for one 32-bit word would probably require on the order of 100 4-LUTs.

A round of Rijndael requires 8 BlockRAMs to store the S-boxes for the byte substitution[10], no area for the row shifting operation, 400 LUTs for the 4 column mixes, 128 LUTs for the key xors and the bypassing of the final column mix, 128 LUTs for the input subkey addition and pipeline MUXes, and 128 LUTs to store the subkeys.

The net result is probably 780 LUTs and 8 BlockRAMs for a single round implementation. With a critical path of 1 memory access, 3 LUTs for the column mixing, and one for the round key addition, a one or two cycle latency is reasonable for a round. With only 10 rounds of encryption, this results in an incredibly low 20 cycles of latency, with a block every 10 cycles.

Rijndael performs a greater number of rounds when used with a larger subkey. This would not affect the area required but would increase the latency and reduce the bandwidth. With 2 clock cycles for each round, it is straightforward to extrapolate the cost of a larger subkey.

4.5 Serpent

Serpent's operations, being very DES-like, map extremely well into hardware. The choice of 4 input, 4 output S-boxes allow each S-box to occupy only 4 4-LUTs, while XORs are very inexpensive, and constant rotations and permutations are free. However, although the algorithm is very fast, a considerable amount of area is required for the S-boxes which make Serpent surprisingly costly in hardware, even though its basic operations are very inexpensive.

4.5.1 Serpent 8-round, 8-slow

Due to the nature of Serpent's S-box use, the sweet spot for a serpent implementation is to unroll 8 rounds. The initial and final permutations require only wiring, not lookup tables, so the entire cost is in the encryption core.

A single round requires 128 LUTs for the key XORs, 128 LUTs to store the subkeys for the round, 128 LUTs for the S-boxes, and 160 LUTs for linear transformation, for a total of 544 LUTs for a single round. In a pipeline, a savings of 64 LUTs/round could be achieved by combining two of the key XORs with the linear transformation from the previous round, at the cost of some design complexity. For an 8 round pipeline, the total comes to 3800 LUTs for the entire pipeline.

[10] There is some wasted memory here due to the size of the BlockRAMs. Only half of the bits are actually used, which indicates that in a technology where the 8x8 S-boxes are directly implemented the area occupied would be smaller

Since each round consists only of bitwise operations and fixed rotations with a critical path of only 5 LUT evaluations, it should be pipelineable with only one cycle/round. It may even be possible to complete 1.5 to 2 rounds in a single cycle, reducing the latency further, since this critical path is so short. Thus, the 8 round pipeline would be run 8 slow, producing a result every 4 cycles, with a low latency of 32 cycles to encrypt a single block.

4.5.2 Serpent single round

A single round implementation would still need to implement all possible S-Boxes, a wide muxing step to combine the results would best be implemented with tristate buffers. Thus, 1024 LUTs would be required for the S-Boxes, 256 LUTs to store the round subkeys, 128 LUTs for the key XOR, and 160 for the linear transform. The resulting single-round implementation would require 1600 LUTs. This also introduces one more evaluation (the muxing of the S-boxes to select the correct one) into the critical path.

If pipelined at the same rate as the 8-round version, this would produce a result every 32 clock cycles, with an identical latency of 32 cycles. Since this only represents a 40% savings in area but an 8-fold reduction in bandwidth, this is not a beneficial tradeoff in most cases.

4.6 Twofish

Twofish works well in hardware without requiring memory to implement S-boxes. Though it is not the fastest or the most compact, it is reasonably small and has other advantages, including a nice area/performance tradeoff and the ability to perform encryption and decryption with a slightly modified pipeline.

The building block of Twofish, the h function, maps reasonably well to FPGA logic. Each q permutation requires 24 LUTs to implement, integrated with the S-box key XORing, for a total of 288 LUTs. The critical path is 12 LUT evaluations, short enough to expect to implement in a single cycle.

The MDS Galois matrix multiplication also maps very well. [3] shows how the multiplication by **0x5b** can be implemented in 8 LUTs, and the multiplication by **0xEF** requires 9 LUTs. It requires a further 8 LUTs to add each output together. The net result is that the matrix requires 135 LUTs to compute, with a critical path of 3 LUTs, allowing it to be combined with the PHT.

4.6.1 Twofish single round

A single round would require 846 LUTs for the two h functions, another 64 LUTs for the PHT, 64 LUTs to store the subkeys, 64 LUTs for the subkey addition, and 64 for the subkey XORing. A final 256 LUTs are required for the whitening steps, resulting in roughly 1360 LUTs for the entire pipeline.

A reasonable expectation would be for this round to take 3 cycles, one for the S-boxes, one for the MDS and PHT, and one for the key addition and XOR[11]. Such a pipeline would take 48 cycles to encrypt a single block, producing a block every 16 cycles.

4.6.2 Twofish folded

Like RC6, the symmetries in Twofish allow the pipeline to be folded in half. This would require an additional cycle to do the PHT, because the MDS would need to be split out, as well as additional logic for the PHT operation. This would require 423 LUTs for the h function, 64 LUTs for the PHT[12], 64 LUTs to store the subkeys, and 64 LUTs to perform

[11] Carries on FPGAs tend to propagate faster then the sum, but if it is necessary to develop a 3 stage pipeline, it might be best to place the pipeline in the middle of the carry of the PHT and key addition, so that the first cycle does the low 16 bits of the PHT and the key addition, and the second cycle does the high bits and the XOR operation

[12] for a 32 bit adder and the additional logic to shift or not and to select the proper input

Algorithm	Implementation	Latency (cycles)	Bandwidth (subkey sets/cycle)	Size (4-LUTs)	Size (BlockRAM)
MARS	New microcoded datapath	270	1/270	300	8
	Existing datpath modified	270	1/270	50	0
RC6	Specialized datapath	264	1/264	290	0
Rijndael	New specialized datapath	36	1/36	128	2
	Shared S-boxes	36	1/36	160	0
Serpent	8 slow, 8 round	32	1/4	2060	0
	2 slow, 2 round	32	1/16	1500	0
Twofish	Shared H-func	20	1/20	512	0
	Separate H-func	4	1/4	1260	0

Figure 2: Comparative performance and cost of subkey generation

the feistel network XOR and to rotate the output if necessary. 128 LUTs would still be needed for each of the whitening steps. It would also require an additional cycle for the PHT, in order to delay the proper element.

Such a pipeline would require roughly 870 LUTs and would require 4 cycles to complete each block, increasing the latency to 64 cycles, and reducing the bandwidth to one block every 32 cycles.

5 Subkey generation

Although subkey generation is not always on the critical path, it is may be necessary to do the subkey expansion within the device, often as a microcoded datapath or customized logic. Some applications, like point-of-sale terminals, may rarely or ever need to change their keys, in which case subkey generation isn't a priority and can be performed external to the device.

Applications such as an encrypting packet router or disk controller may require changing subkeys on a packet-by-packet or block-by-block basis. In such applications, the key setup time and parallelism may prove to be the critical factor. An important consideration for hardware implementations is how agile the key scheduling is. Being able to pipeline subkey generation at the same rate as encryption allows subkeys to be generated concurrent to encryption.

Note, though, that Rijndael and Serpent allow concurrent keyscheduling only in the encryption direction, not for decryption. These ciphers require some additional buffering for the expanded subkeys for decryption, which would make decryption latency for a changed key to be different than the encryption latency for a changed key.

The ideal case, which only occurs in Twofish, is subkeys which can be generated independently. This allows encryption and decryption subkeys to be generated on the fly regardless of whether the data is being encrypted or decrypted. This is a great advantage for devices which need to encrypt and decrypt a large number of differently keyed blocks.

In general, the datapath will only be described for a keysize of 128 bytes, if there is a significant difference in the pipeline structure for different key sizes.

Both MARS and RC6 have considerably slower subkey generation when compared with the other candidates. Neither can be effectively pipelined or accelerated, and any attempt to simultaneously produce multiple subkeys for different initial keys requires duplication of the subkey-creating hardware.

Rijndael's subkey generation is considerably shorter and takes up a small amount of area.

Although it can not be pipelined, it is small enough to duplicate if subkeys are changed often. Creating the Serpent subkeys, on the other hand, favor a heavily pipelined design due to the comparatively high cost of all the S-boxes.

Twofish's key generation can share hardware with the encryption pipeline, if a low cost implementation is required. Alternatively, it may contain it's own copy of the S-box logic and generate the subkeys concurrently with encryption, essentially eliminating all the latency involved in subkey creation.

5.1 MARS

The MARS subkey generation is best implemented in a custom microcoded datapath. If such a datapath is used for encryption, the incremental cost of subkey generation is minor, just a fair amount of expanded code with all indexes recalculated. The only addition would be a logical structure to compute M_n requiring some 100 LUTs to accomplish. If a microcoded datapath is not used, essentially the full microcoded datapath from the encryption description (sans multiplier), would be necessary, roughly 300 LUTs and 8 BlockRAMs, and roughly 270 cycles to generate the subkeys.

5.2 RC6

The RC6 subkey generation is probably best implemented with a custom datapath, using 2 BlockRAMs to store the subkeys during computation. Since the number of user key blocks is rather small, 32 LUTs used as a small RAM is sufficient. 2, 32 bit registers can store A and B, with 32 LUTs for a dedicated adder to always compute $A + B$. The only additional logic to calculate A is 2 adders, one to generate the initial value of the S array, and the second to add the current value of the S array to $A + B$, 64 LUTs in all. For updating B, this requires 160 LUTs for the rotation and 32 LUTs for another adder. Thus, the total datapath would occupy 290 LUTs.

The control logic for this structure consists only of a couple of counters and some simple state for the state machine, so it should not require significant resources.

It should be reasonable to update A in 1 cycle as it only requires 3 additions or two additions plus a memory lookup, and a constant rotation. Similarly, B should be computable in a single cycle as well. Thus, for 20 round RC6, this datapath requires 132 executions, for 264 cycles to generate the subkeys.

5.3 Rijndael

Rijndael's subkey generation is very compact. It can only produce four bytes per cycle as each word is dependent on the previous word, so an implementation which changes keys often would be still dominated by the latency of subkey generation.

Subkey generation requires 4 copies of the S-boxes in 2 BlockRAMs (either shared with the encryption pipeline or independent), enough buffering for 128b with a 128b key, 32 LUTs for the Rcon table, and 32 LUTs for the various XORs and selections. Since the buffering is dominant, the total would probably require 128 LUTs, as the flip flops end up dominating the cost. Each subkey word could be generated in a single cycle, requiring 38 cycles to generate all the subkeys.

5.4 Serpent

Just as the best Serpent implementation is an 8 round, 8 slow pipeline, the same holds for the subkey generation. Since the structure is very similar to the round itself, the same techniques can be used. It requires 64 LUTs to calculate the XORs for each of the 4 subkeys generated for each round, another 128 LUTs for the sbox substitution, and 32 LUTs for calculating the index, for 224 LUTs for each round. At 8 rounds, this comes to 1800 LUTs plus another 260 LUTs for the MUXes at the end of the pipeline, for a total of 2060 LUTs.

A more compact, 2 round design would still require 128 LUTs for the XORs, 1024 LUTs for the S-boxes with the results muxed by tristate buffers, 64 LUTs for the indexes, and 256 LUTs for the MUXes at the start of the pipeline. This would total to roughly 1500 LUTs, while still only requiring 32 cycles to generate a full set of subkeys. Although it might be possible to reuse the S-boxes from the encryption pipeline, the additional muxing would probably swamp most of the savings achieved by this reuse unless only a single-round serpent implementatino is used.

5.5 Twofish

The key generation in Twofish occurs in two parts, the first generating the two keys for the S-boxes and the second generating the round keys. The S-Box subkeys require a constant GF(2^8) matrix multiplication. Using specialization, assuming an average of 8 LUTs/constant, 256 LUTs are required to generate the subterms, and another 96 LUTs are required to perform the XORs to generate the sbox subkeys.

For implementations where subkey generation is not in the critical path one can use the S-boxes from the encryption pipeline. The modifications to the existing pipeline would add 64 LUTs to mux the inputs into the S-boxes, 64 LUTs to mux the S-box subkeys between the encryption subkeys and the input keys, and another 32 LUTs to modify the PHT, for a total of 160 LUTs, a very small addition to the pipeline. This approach would require a total of 512 LUTs of datapath to generate the subkeys and 20 cycles to generate the complete set of subkeys.

A separate round subkey datapath could be implemented, requiring an additional copy of the 2 H-functions and a PHT (910 LUTs). This would require a total area of 1260 LUTs. This is comparable to the cost of the encryption pipeline, but has the advantage that subkeys can be generated on the fly concurrently with encryption, except for those subkeys required for the input and output whitening. This allows a hardware implementation of twofish to operate at almost maximum bandwidth while able to change subkeys on a block by block basis, and to shift between encryption and decryption at will. This has the effect of reducing the key setup time to only the 4 cycles needed to generate the input and output whitening subkeys.

6 Other implementations

Twofish and Serpent have hardware implementations[3] [5] reported in the literature which can be used to help calibrate the quality of our estimates. Both implementations used HDL synthesis, which hurts performance but does not significantly affect the area required.

The Twofish implementation in [3] requires roughly 900 Xilinx 4000 CLBs, or 1800 LUTs, with the hardware for the round itself requiring roughly 1400 LUTs. A pipelined version used 7 cycles/round, running at 35 MHz. Three considerations reduced their performance: the implementation overhead of VHDL, routing congestion and tools, and an older generation FPGA.

The area numbers for this implementation are very close to the estimates for Twofish, a very good sign. Also, the performance degradation present in the HDL version is expected. HDL synthesis[13] techniques tend to produce significantly lower performing designs[14], and the Xilinx 4000 series is also significantly slower then the current generation of Xilinx FPGAs.

[13] This is where the logic is described in a High level Description Language and then compiled to form the actual circuitry of the implementation, as opposed to a lower level approach of a had specified and hand placed datapath which is assumed in the estimates.

[14] There are two factors involved: HDL synthesis on a design like Twofish is usually constructed without detailed placements for the individual modules of the datapath, and the place-and-route tools are not intelligent about placing or reconstructing datapaths in designs.

The Serpent implementation [5] is unfortunately harder to use as a calibration. It required 18,000 LUTs at 37 MHz for a fully unrolled, pipelined (1 stage/round) version, 15,000 LUTs at 13 MHz for an unpipelined, 8 round version, and 11,000 LUTs at 15 MHz for a single round when implemented in a Virtex 1000. The performance numbers are very good, and although some improvement may be achieved by a manually layed-out design, the nature of serpent doesn't have heavy datapath regularity to exploit.

The single and eight round versions can not be used to calibrate the area estimates, as the design used flipflops and MUXes for subkey storage, instead of the luts-as-memory ability present in the Virtex. Furthermore, the single round implementation used MUXes instead of the internal tristate lines to mux the S-boxes, a serious inefficiency in the implementation.

The best mechanism for attempting to calibrate area is to quadruple the area estimate for an eight round version of serpent, as a first approximation. With 15,000 LUTs for the estimated area, and 18,000 LUTs for the HDL implementation, the comparison is pretty close. The additional area for the HDL version undoubtedly includes the logic for setting the subkeys and performing I/O, while the estimate in this paper only considers the cryptographic core.

7 Conclusions and Lessons Learned

Both Rijndael and Twofish are very amenable to hardware implementations. Rijndael is the fastest, with a great degree of parallelism and very quick operations, but area requirements increase substantially if encryption and decryption is required in the same device. Although Twofish is somewhat slower, there is an excellent degree of flexibility in subkey generation and in area/performance tradeoffs.

The numerous, large S-boxes are one of the features which greatly cripple MARS hardware implementations. Having to implement 9 large 32bit S-boxes to create a single C-slow pipeline impose a significant cost on any implementation. Also, the heterogeneous round types cause a significant area penalty when compared to other implementations. The use of both S-boxes and multiplication further compounds the cost, requiring both considerable storage and considerable logic to implement.

The subkey generation for both MARS and RC6 have serial steps which require all subkeys to be modified several times. This causes subkey generation to be very slow in hardware, a significant defect when dealing with applications which require rapidly changing subkeys.

Serpent ends up being surprisingly awkward, mostly due to the large number of S-boxes required. It takes 1024 LUTs just to store all the S-boxes. Although the performance is excellent, the bandwidth quickly drops for smaller implementations and the area/performance suffers greatly.

Similarly, two operations which are cheap software, multiplication and rotation, end up being comparatively expensive in hardware. A multiplier occupies much more logic then an addition in hardware, and large multiplier are much costlier[15]. Similarly, variable rotations are much more expensive in hardware when compared to constant rotations, XORs, or additions.

Independently generated subkeys such as those in Twofish offer a great benefit for some applications, as this allows almost complete hiding of the subkey generation time. This property allows a hardware implementation to almost completely overlap subkey generation with encryption and can remove the need for any expanded subkey storage.

[15] As an example, a $32x32$ modulo 2^{32} multiplier is four times the area of a $16x16$ modulo 2^{16} multiplier.

8 Acknowledgments

Many thanks to David Wagner for explaining the design decisions and operations of various aspects of the ciphers and to Eylon Caspi for his capable editing.

References

[1] Anderson, Biham, and Knudsen, "Serpent: A Proposal for the Advanced Encryption Standard", *http://csrc.nist.gov/encryption/aes/round2/AESAlgs/Serpent/Serpent.pdf*

[2] Burnwick *et al*, "The MARS encryption algorithm", *http://csrc.nist.gov/encryption/aes/round2/AESAlgs/MARS/mars-int.pdf*

[3] Chodowiec and Gaj, "Implementation of the Twofish Cypher Using FPGA Devices", George Mason University Techinical Report, *http://www.counterpane.com/twofish-fpga.html*

[4] Daemen and Rijmen, "AES Proposal: Rijndael", *http://csrc.nist.gov/encryption/aes/round2/AESAlgs/Rijndael/Rijndael.pdf*

[5] Elbirt and Parr, "An FPGA Implementation and Performance Evaluation of the Serpent Block Ciphen", in *ACM/SIGDA International Symposium on Field Programmable Gate Arrays (FPGA)*, February, 2000.

[6] Rivest, Robshaw, Sidney, and Yin, "The RC6 Block Cypher", *http://csrc.nist.gov/encryption/aes/round2/AESAlgs/RC6/cipher.pdf*

[7] Schneier *et al*, "Twofish: A 128-Bit Block Cypher", *http://csrc.nist.gov/encryption/aes/round2/AESAlgs/Twofish/Twofish.pdf*

[8] Triscend Inc, "Triscend E5 Configurable System-on-Chip Family", *http://www.triscend.com/products/dse5csoc.pdf*

[9] Xilinx Inc, "Virtex 2.5V Field Programmable Gate Arrays", *http://www.xilinx.com/partinfo/ds003.pdf*

Comparison of the hardware performance of the AES candidates using reconfigurable hardware

Kris Gaj and Pawel Chodowiec
George Mason University
kgaj@gmu.edu, pchodowi@gmu.edu

Abstract

The results of implementations of all five AES finalists using Xilinx Field Programmable Gate Arrays are presented and analyzed. Performance of four alternative hardware architectures is discussed and compared. The AES candidates are divided into three classes depending on their hardware performance characteristics. Recommendation regarding the optimum choice of the algorithms for AES is provided.

1. Introduction

Hardware implementations of cryptography will thrive in the new century because of the growing requirements for high-speed, high-volume secure communications combined with physical security. In the presence of no major breakthroughs in cryptanalysis of the AES candidates, and relatively inconclusive results of their software performance evaluation [NBD+99, SKW+99], the comparison of the hardware performance of the AES algorithms may provide a major indicator for a final decision regarding the new standard.

Very few results regarding hardware implementations of the AES candidates have been published so far. Original documentation provided by designers of the submitted algorithms contains typically only rough estimates of the hardware performance [BCD+98, RRS+98, SKW+98]. Additionally, these estimates are very difficult to compare among each other because of large differences in assumptions regarding the technology, and because of different architecture choices. The results of actual implementations of individual algorithms, published recently by independent researchers [EP99, RH99], provide only a very fragmentary knowledge, not suitable for reliable comparison.

This situation will be certainly remedied by the publication of the NSA findings regarding hardware performance of the AES candidates. Nevertheless, the NSA evaluation plan [NSA98] targets only implementations using *semi-custom Application Specific Integrated Circuits* (ASICs), providing no data regarding other technologies. In this article, we focus on comparing AES candidates using an alternative hardware technology based on Field Programmable Gate Arrays (FPGAs). This technology, referred to as *reconfigurable hardware*, offers many advantages for future vendors and users of cryptographic equipment. It assures a short time to the market, high flexibility (including a capability for frequent modifications of hardware), low development costs, and low cost of the final product - the result of the algorithm agility - capability to use the same integrated circuit with time sharing for the execution of various secret-key and public-key algorithms. Our comparison supplements the NSA effort by covering the second primary way of implementing cryptographic algorithms in hardware.

2. Reconfigurable hardware

2.1 Operation and internal structure of an FPGA device

Field Programmable Gate Array (FPGA) is an integrated circuit that can be bought off the shelf and reconfigured by designers themselves. With each reconfiguration, which takes only a fraction of a second, an integrated circuit can perform a completely different function. FPGA consists of thousands of universal building blocks, known as *Configurable Logic Blocks* (*CLBs*), connected using programmable interconnects, as shown in Fig. 1a. Reconfiguration is able to change a function of each CLB and connections among them, leading to a functionally new digital circuit.

From several FPGA families available on the market, we have chosen for implementing AES candidates two families from Xilinx, Inc.: high performance Virtex family, and a low-cost XC4000 family. Each family consists of several FPGA devices, manufactured in the same technology, covering certain range of maximum circuit sizes.

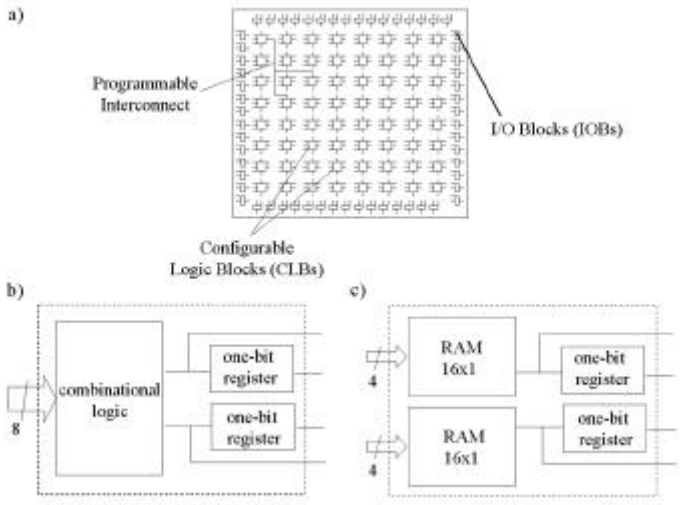


Fig. 1 FPGA device. a) General structure and main components. b) Internal structure of a CLB configured in the logic mode. c) Internal structure of a CLB configured in the memory mode.

A simplified internal structure of a CLB in the XC4000 family, and a *CLB slice* (1/2 of a CLB) in the Virtex family is shown in Figs. 1bc. In the logic mode (Fig. 1b), each of these elementary units contains a small block of combinational logic, implemented using programmable look-up tables, and two one-bit registers. In the memory mode, combinational logic is replaced by two small memories. A CLB in the XC4000 family of FPGA devices and a CLB slice in Virtex are functionally almost identical. Therefore, we will use a number of these elementary units, necessary to build a given circuit, as a measure of the circuit area and cost.

2.2 Advantages of using reconfigurable hardware for comparison of the AES candidates

For implementing cryptography in hardware, FPGAs provide the only major alternative to *custom and semi-custom Application Specific Integrated Circuits* (ASICs), integrated circuits that must be designed all the way from the behavioral description to the physical layout, and sent for an expensive and time-consuming fabrication. The comparison of the AES candidates based on FPGA devices has the following advantages over the comparison based on ASICs:

- Shorter design cycle leading to fully functioning device prototypes.
- Lower cost of the computer-aided design tools, verification, and testing.
- Potential for fast, low-cost multiple reprogramming and experimental testing of a large number of various architectures and revised versions of the same architecture.
- Higher accuracy of comparison: in the absence of the physical design and fabrication, ASIC designs are compared based on inaccurate pre-layout simulations [NSA98]; FPGA designs are compared based on very accurate post-layout simulations and experimental testing.

3. Alternative architectures

3.1 Basic organization of a block cipher implementation

The basic organization of the hardware implementation of a symmetric block cipher is shown in Fig. 2. All five AES candidates investigated in this paper can be implemented using this organization. The organization includes the following units:

a. *Encryption/decryption unit*, used to encipher and decipher input blocks of data.
b. *Key scheduling unit*, used to compute a set of internal cipher keys based on a single external key.
c. *Memory of internal keys*, used to store internal keys computed by the key scheduling unit, or loaded to the integrated circuit through the input interface.
d. *Input interface*, used to load blocks of input data and internal keys to the circuit, and to store input blocks awaiting encryption/decryption.

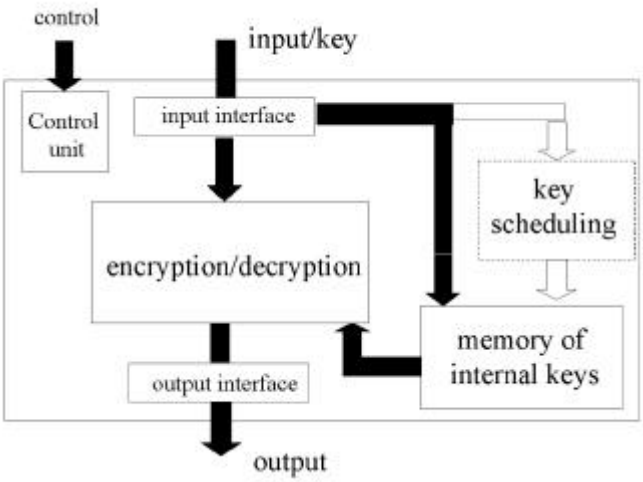


Fig. 2 Block diagram of the hardware implementation of a symmetric-block cipher.

e. *Output interface*, used to temporarily store output from the encryption/decryption unit and send it to the external memory.
f. *Control unit*, used to generate control signals for all other units.

3.2 Feedback vs. non-feedback operating modes

Today's symmetric block ciphers are used in several operating modes. From the point of view of hardware implementations, these modes can be divided into two major categories:

a. *Non-feedback modes*, such as Electronic Code Book mode (ECB), and counter mode.
b. *Feedback modes*, such as Cipher Block Chaining mode (CBC), Cipher Feedback Mode (CFB), and Output Feedback Mode (OFB).

In the non-feedback modes, encryption of each subsequent block of data can be performed independently from processing other blocks. In particular, all blocks can be encrypted in parallel. In the feedback modes, it is not possible to start encrypting the next block of data until encryption of the previous block is completed. As a result, all blocks must be encrypted sequentially, with no capability for parallel processing.

According to current security standards, the encryption of data is performed primarily using feedback modes, such as CBC and CFB. Non-feedback modes, such as ECB, are used primarily to encrypt session keys during key distribution. As a result, using current standards does not permit to fully utilize the performance advantage of the hardware implementations of secret key cryptosystems, based on parallel processing of multiple blocks of data.

3.3 Alternative architectures for the encryption/decryption unit

a. Basic architecture

The basic hardware architecture used to implement an encryption unit of a typical secret-key cipher is shown in Fig. 3a. One round of the cipher is implemented as a combinational logic, and supplemented with a single register and a multiplexer. In the first clock cycle, input block of data is fed to the circuit through the multiplexer, and stored in the register. In each subsequent clock cycle, one round of the cipher is evaluated, the result is fed back to the circuit through the multiplexer, and stored in the register. The number of clock cycles necessary to encrypt a single block of data is equal to the number of cipher rounds, #rounds.

We define the *speed* of the cipher implementation as the number of bits of data encrypted in a unit of time. Speed calculated this way is often referred to as the circuit *throughput*. The speed of the basic architecture, $speed_{ba}$, is given by

$$speed_{ba} = 128/\ \#rounds \cdot clock\_period\ . \qquad (1)$$

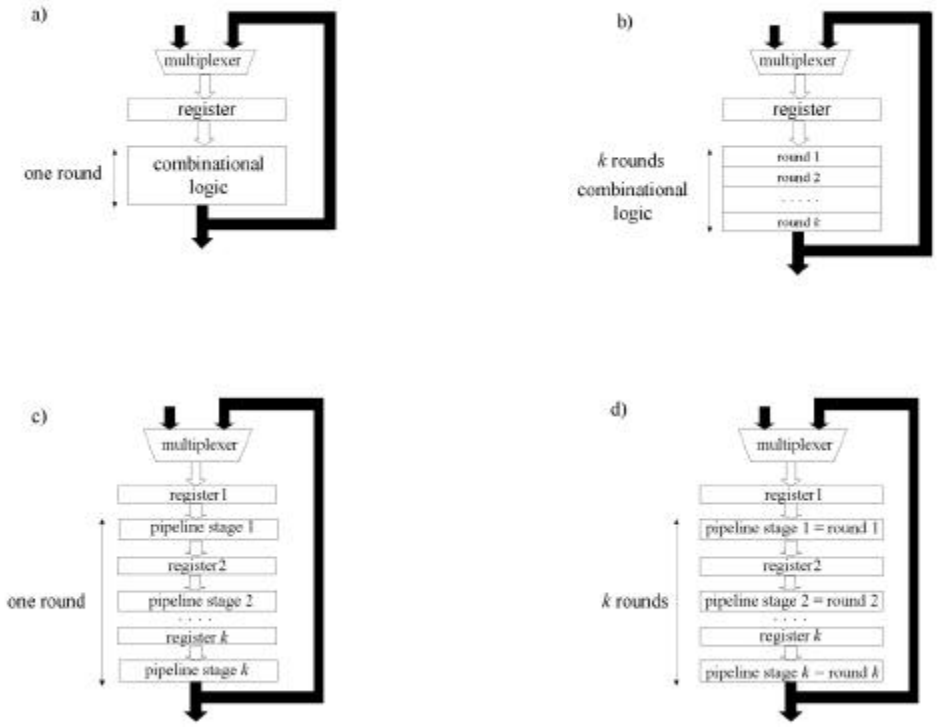


Fig. 3 Four alternative architectures for implementation of an encryption/decryption unit of a block cipher: a) basic architecture, b) architecture with the *k*-round loop unrolling, c) architecture with the *k*-stage inner-round pipelining, d) architecture with the *k*-stage outer-round pipelining.

The basic architecture combines a good speed with the relatively modest area requirements. However there exist several alternative architectures that permit to improve either one or both of these performance measures.

b. Loop unrolling

Architecture with loop unrolling is shown in Fig. 3b. The only difference compared to the basic architecture is that the combinational part of the circuit implements *k* rounds of the cipher, instead of a single round. The maximum value of *k* is equal to the number of cipher rounds. The number of clock cycles necessary to encrypt a single block of data decreases by a factor of *k*. At the same time the minimum clock period increases by a factor slightly smaller than *k*, leading to an overall relatively small increase in the cipher speed, given by

$$speed_{lu}/speed_{ba} = (1 + \tau)/(1+\tau/k), \quad (2)$$

where τ is the ratio of the sum of the multiplexer delay, the register delay and the register setup time to the delay of a single cipher round. This increase in speed is obtained at the cost of the circuit area. Because the combinational part of the circuit constitutes the majority of the circuit area, the total area of the encryption/decryption unit increases almost proportionally to the number of unrolled rounds, *k*. Additionally, the number of internal keys used in a single clock cycle increases by a factor of *k*, which in FPGA implementations typically implies the almost proportional growth in the number of CLBs used to store internal keys.

In summary, loop unrolling enables increasing the circuit speed in both feedback and non-feedback operating modes. Nevertheless this increase is relatively small, and incurs a large area penalty.

c. Inner-round pipelining

Pipelining is a general method of increasing the amount of data processed by a digital circuit in a unit of time. The idea is to introduce evenly spaced extra registers in the middle of the combinational circuit, in such a way that several blocks of data can be processed by the circuit at the same time. Parts of the combinational logic divided by adjacent registers are called pipeline stages (see Fig. 3c). In each clock cycle the partially processed data block moves to the next pipeline stage. Its place is taken by the subsequent data block. This way, a pipelined circuit can encrypt simultaneously as many blocks of data, as the number of pipeline stages it contains.

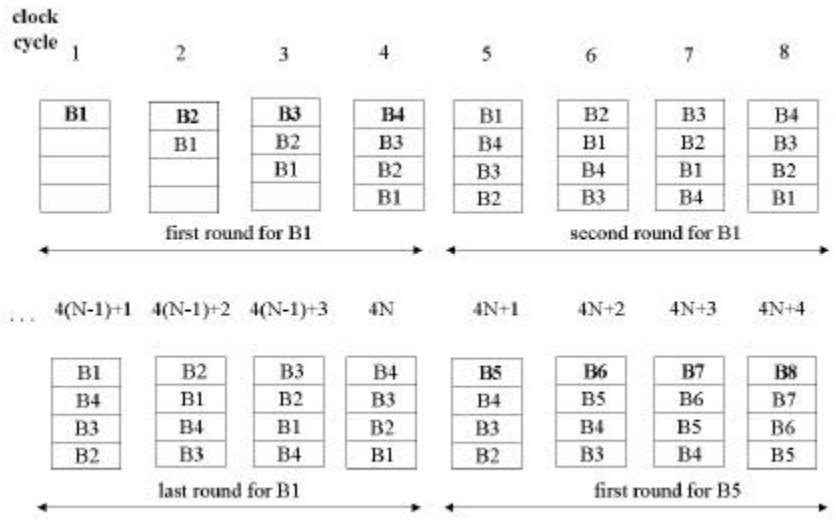


Fig. 4 Operation of the architecture with 4-stage inner-round pipelining for an N-round cipher.

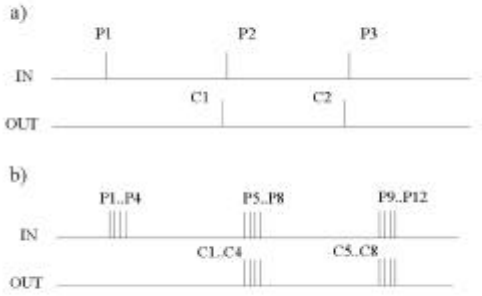


Fig. 5 Timing of input and output blocks in a) basic architecture, b) architecture with a 4-stage inner-round pipelining.

The flow of data through the pipeline during encryption is shown in Fig. 4. The number of pipeline stages in this example is four. During the first four clock cycles four subsequent blocks of data enter the pipeline. In the subsequent clock cycles, these blocks circulate in the pipeline. Each four clock cycles correspond to a single cipher round. In the cycle number 4·#rounds+1, the first block, B1, leaves the pipeline, and the fifth block, B5, is introduced to the empty pipeline stage. In the following three clock cycles, blocks B2, B3, and B4, leave the pipeline, substituted by blocks B6, B7, and B8. The timing diagram of the input and output of the circuit is shown in Fig. 5b. Speed of the circuit, expressed as the number of bits processed by the circuit in a unit of time is given by

$$\text{speed} = 128/\ \#\text{rounds} \cdot \text{reduced\_clock\_period} \tag{3}$$

where reduced_clock_period is a minimum clock period after pipelining.

The dependence between the cipher speed-up resulting from the inner-round pipelining and the number of evenly spaced pipeline stages is shown in Fig. 6. There exists a maximum number of pipeline stages that still improves the circuit throughput. Adding additional registers will not affect the throughput. The maximum number of pipeline stages is determined by the delay of the largest indivisible combinational portion of the circuit. For majority of ciphers it is difficult to divide the cipher round into combinational stages with equal delays (especially, when the circuit is described in a high-level hardware description language, such as VHDL),

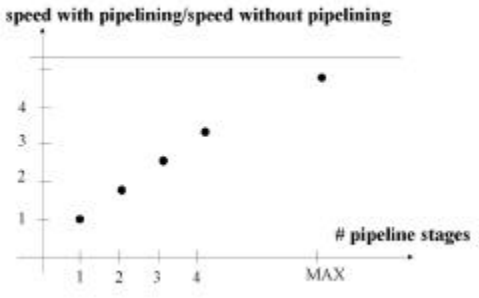


Fig. 6 Speed of the architecture with *k*-round inner-round pipelining as a function of the number of evenly spaced pipeline stages.

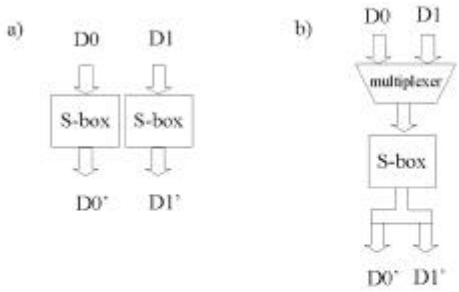


Fig. 7 Resource sharing of an S-box. a) basic operation of two parallel S-boxes, b) operation with resource sharing.

which further limits the circuit speed-up. Area of the circuit with inner-round pipelining increases only by a small percentage (area of a single 128-bit register) with each additional pipeline stage. This is especially true for FPGA circuits, in which CLBs used to implement combinational logic often contain registers not utilized in the non-pipelined implementation.

d. Outer-round pipelining

Outer-round pipelining is created by loop unrolling followed by introducing extra registers between parts of the combinational logic corresponding to each cipher round, as shown in Fig. 3d. The number of unrolled loops *k* is typically a divisor of the total number of cipher rounds, #rounds.

Area of the encryption unit with outer-round pipelining is directly proportional to the number of pipeline stages *k*. In the non-feedback cipher modes, such as ECB, the speed (throughput) of the cipher increases proportionally to the number of pipeline stages, *k*. Therefore, the outer-round pipelining enables to directly trade circuit speed with circuit area. In the feedback cipher modes, the speed of the cipher remains independent of the number of outer pipeline stages, and therefore, this kind of pipelining is not recommended for these modes.

e. Resource sharing

For some ciphers, it is possible to further decrease circuit area by time sharing of certain resources (e.g., function *h* in Twofish, 4x4 S-boxes in Serpent, 8x32 S-boxes S0, S1 in the mixing transformation of Mars, multiplication units in RC6). This is accomplished by using the same functional unit to process two (or more) parts of the data block in different clock cycles, as shown in Fig. 7b. In Fig. 7a, two parts of the data block, D0 and D1, are processed in parallel, using two independent S-boxes. In Fig. 7b, a single S-box is used to process two parts of the data block sequentially, during two subsequent clock cycles.

The use of resource sharing in real life implementations is expected to be limited, because

- Gain in the circuit area is always smaller than the loss in the circuit speed.
- The amount of area used by a basic implementation of a symmetric cipher is typically already quite small.

3.4. Choice of the figure of merit

The choice of a single figure of merit is difficult, because the optimization criteria may vary depending on the application. In our comparison, we took into account three basic figures of merit: maximum speed (throughput), minimum area, and the maximum speed/area ratio.

Optimization for maximum speed will be done in applications where communication requirements force the use of a very high speed encryption, and/or the cost of the cryptographic hardware constitutes only a small portion of the entire system. Examples of such applications include ATM and ISDN switches, Virtual Private

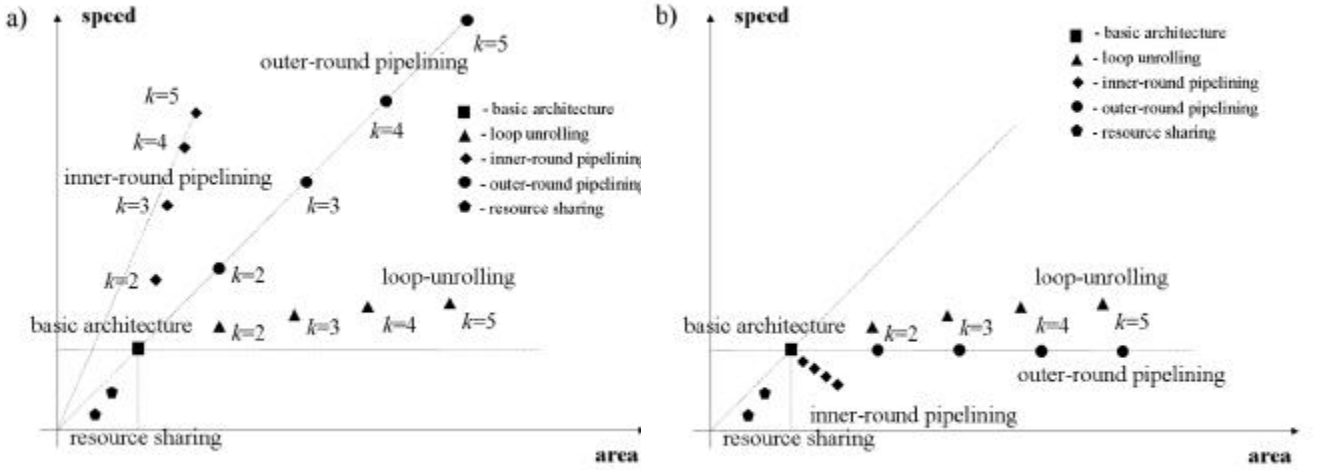


Fig. 8 Hardware performance of various alternative architectures in a) non-feedback cipher modes, such as ECB and counter mode, b) feedback cipher modes, such as CBC, CFB, and OFB.

Network routers and firewalls, WWW and database servers. In such applications, it may be justified to trade the cost of the cryptographic hardware (proportional to the circuit area) for greater speed.

In the second class of applications, the designer's goal is to obtain the maximum speed, assuming a given limit on the circuit area (cost). In such situations, the more appropriate figure of merit is the speed/area ratio. This figure of merit is particularly appropriate for non-feedback cipher modes, which enable one to directly trade circuit area for speed by using the outer-round pipelining, as shown in Fig. 8a. The examples of cost critical applications of cryptography include pagers, digital video recorders, and PCMCIA cards.

Applications that require optimization for minimum area include smart cards, embedded systems, and cellular phones. As the basic architecture may be still too big for such applications, they may enforce resource sharing. Taking into account the size and power limitations, these applications will be typically implemented using custom ASICs, not FPGAs.

3.5 Comparison of various architectures

Dependencies between the speed and the area of the encryption/decryption unit of a block cipher, for architectures discussed in section 3.3, are shown in Fig. 8.

a. Non-feedback modes

For non-feedback modes, the best speed/area ratio can be obtained by using inner-round pipelining with the maximum number of pipeline stages that still increases circuit clock frequency, as shown in Fig. 8a. The largest possible speed can be obtained by combining inner-round pipelining with outer-round pipelining. The only limit on the circuit speed is imposed in this case by the maximum circuit area (cost) and/or the maximum number of the outer-round pipeline stages (equal to the number of the cipher rounds). The smallest possible area can be obtained using the basic architecture with resource sharing.

b. Feedback-modes

For feedback modes, the basic architecture offers the best value of the ratio speed/area, as shown in Fig. 8b. Larger speed can only be obtained using loop unrolling, at the cost of a very significant increase in the circuit area (cost). Smaller area can only be obtained using resource sharing, at the cost of the significant reduction in the circuit speed.

Outer-round pipelining is inefficient in these modes, as it does not increase circuit speed, and significantly increases circuit area. Inner-round pipelining decreases speed, and increases circuit area. As a result, neither type of pipelining should be used in these operating modes.

4. Assumptions

4.1 Primary assumptions

The following tentative assumptions have been made in order to simplify the task of comparing AES candidates:

a. Key size 128 bits.

Our implementations are intended to support only one key size, 128 bits. Other key sizes required by AES (192 and 256 bits), or supported by a particular algorithm will be added in the future.

b. No key scheduling unit.

Our implementations do not support the on-chip generation of internal keys from a single external key. Instead, our implementations include a memory of internal keys loaded with the keys generated externally, and the circuitry necessary to distribute these keys from the memory to the encryption/decryption unit.

c. *Block size 128 bits.*

Only one input/output block size, 128 bits, has been considered, even if the given AES candidate supports other block sizes.

d. Basic architecture

The encryption part of all AES candidates has been implemented using basic architecture shown in Fig. 3a. This architecture has been chosen for the following reasons:

* As shown in Fig. 8b, the basic architecture assures the maximum *speed/area* ratio for feedback operating modes (CBC, CFB), now commonly used for bulk data encryption. It also guarantees near optimum speed, and near optimum area for these operating modes.
* The basic architecture is relatively easy to implement in a similar way for all AES candidates, which supports fair comparison. For architectures with inner-round pipelining, it is relatively difficult to determine and implement the maximum number of pipeline stages that still increases circuit speed and speed/area ratio.
* The implementations of the basic architecture exemplify larger differences among five AES algorithms compared to the architectures with inner-round pipelining. Inner-round pipelining permits decreasing the differences in speed among various ciphers because ciphers with longer critical path (lower speed) may be sped up by a larger factor by introducing proportionally more pipeline stages.
* Based on the performance measures for basic architecture, it is possible to derive analytically *approximate* formulas for parameters of more complex architectures, including architectures with outer-round pipelining (near proportional scaling of both area and speed), loop-unrolling (see formula (2)), and inner-round pipelining (see formula (3) and Fig. 6). Nevertheless, these formulas should be treated only as a first approximation, and the more detailed comparison requires the actual implementation of all ciphers using alternative architectures. Only such implementations may take into account the exact structure of all ciphers, limitations imposed by the FPGA architecture and the design entry method (e.g., VHDL description), and the optimization capabilities of the FPGA computer-aided design tools.

e. Resource sharing between the encryption and decryption part

In order to minimize circuit area, it was assumed that the encryption and decryption parts share as many resources as possible by the given cipher type. The effort was made to maximally decrease the effect of resource sharing on the speed of encryption and decryption.

4.2 Deviations from the basic architecture

Three ciphers, Twofish, RC6, and Rijndael, have been implemented using exactly the basic architecture shown in Fig. 3a. This was possible because all rounds of these ciphers perform exactly the same operation. For the remaining two ciphers, Serpent and Mars, this condition is not fulfilled, and as a result, small deviations from the basic architecture appeared to be necessary.

Serpent consists of 8 different rounds repeated 4 times. Therefore, it is advantageous to treat 8 official cipher rounds as a single *implementation round*, and assume that the cipher has 4 rounds. This way, 8 official cipher rounds are implemented in the basic architecture as a combinational logic. This implementation guarantees the maximum speed/area ratio typical for the basic architecture.

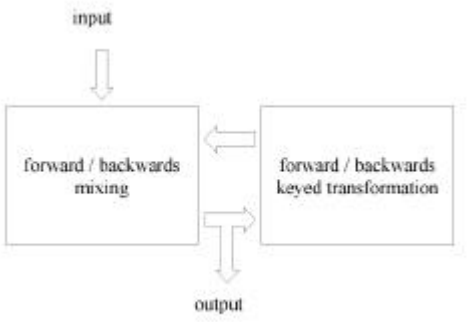


Fig. 9 Deviation from the basic architecture in Mars.

In Mars, there exist four different kinds of rounds, each repeated 8 times: forward mixing, forward keyed transformation, backwards keyed transformation, and backwards mixing. It is possible to implement forward and backwards mixing using the same functional unit; the same holds for the forward and backwards keyed transformation. The structure of the mixing transformation and the keyed transformation are significantly different, and as a result they must be implemented using separate units, as shown in Fig. 9. Both of these units have an internal structure that corresponds to the basic architecture (multiplexer + register + combinational logic). Additionally, both units share the look-up table implementing two 8x32 S-boxes.

5. Results

5.1 Results for the Virtex family

The results of implementing AES candidates, according to the assumptions summarized in section 4, using the largest currently available Xilinx Virtex device, XCV1000BG560-6, are summarized in Fig. 10. For comparison, the results of implementing the current NIST standard, Triple DES, are also provided It should be stressed that all results come either from simulation or from reports generated by Xilinx tools, and have not as yet been confirmed experimentally. The details of all implementations, including the detailed block diagrams, and the description of simulation and test experiments will be provided in the technical report available at the AES conference [CG00]. Part of this report, describing Twofish, is already available on the web [CG99].

Implementations of all ciphers take from 9% (for Twofish) to 38% (for Serpent) of the total number of 12288 CLB slices available in the Virtex device used in our designs. It means that less expensive Virtex devices could be used for all implementations. Additionally, the key scheduling unit can be easily implemented within the same device as the encryption/decryption unit.

5.2 Results for the XC4000 family

For the low-cost, medium-size family of Xilinx FPGA devices, XC4000, only two ciphers, Twofish and RC6, were able to fit within the largest device from this family. The relative performance of these ciphers is similar to the relative performance in Virtex implementations. It is interesting to notice that for the two different FPGA devices from this family, the smaller one guarantees the higher speed.

Cipher	**Speed [Mbit/s]**		**Area [CLBs]**		**Speed/Area [kbit/s·CLB]**	
	4028/4036	4085	4028/4036	4085	4028/4036	4085
Twofish	90.9	89.2	907	907	100.2	98.3
RC6	45.9	43.1	1222	1222	37.6	35.3

Table I. Results of implementing Twofish and RC6 using the largest available FPGA device from the XC4000XL family, XC4085XL, and the largest device fitting the implementation of the respective cipher, i.e., XC4028XL for Twofish, and XC4036XL for RC6.

5.3 Resource sharing between encryption and decryption

The amount of resource sharing between encryption and decryption is considerably different for various AES candidates, depending on the type of the cipher. Resource sharing is close to 100% for Feistel ciphers and modified Feistel ciphers, and close to zero for S-P networks. The level of resource sharing can be described by the amount and type of the extra logic that must be added to the circuit implementing encryption, so that the modified circuit can perform both encryption and decryption, as shown in Table II.

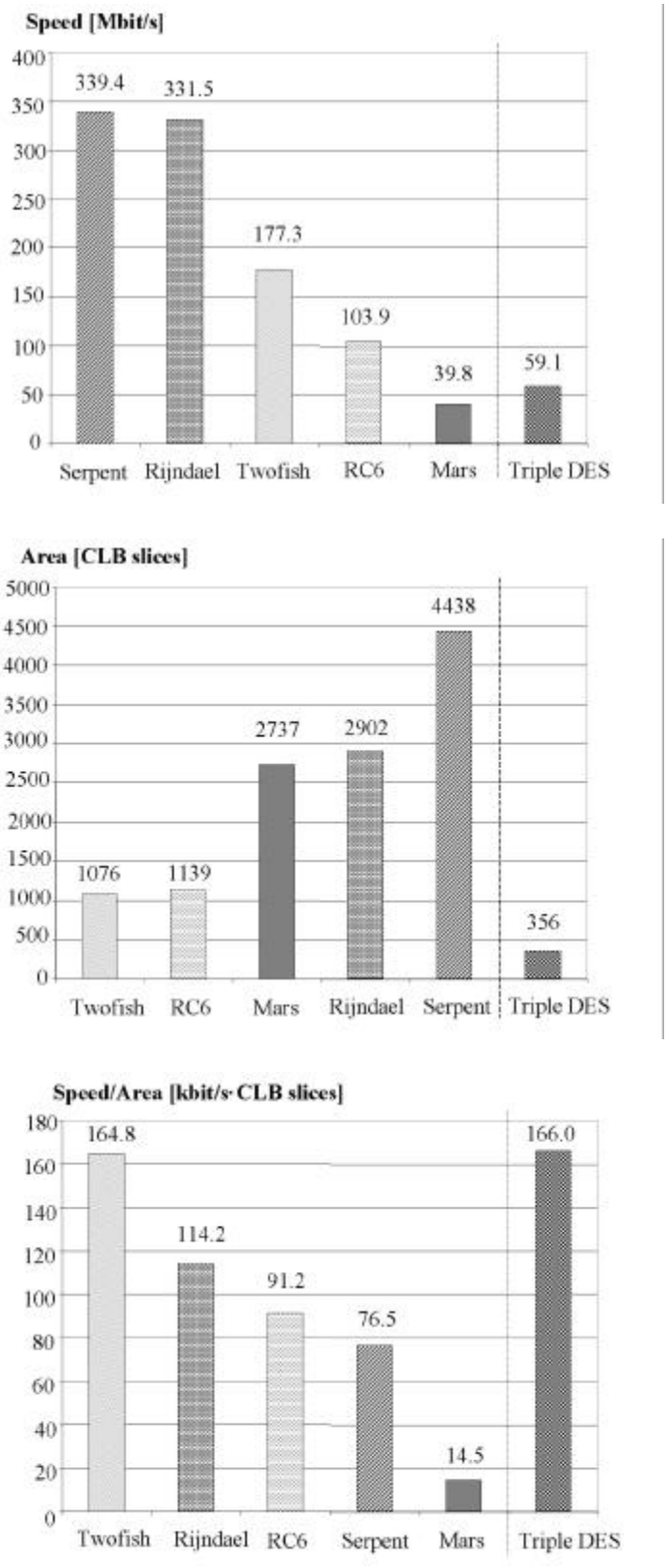


Fig. 10 Results of implementing AES candidates using Xilinx Virtex FPGA devices.

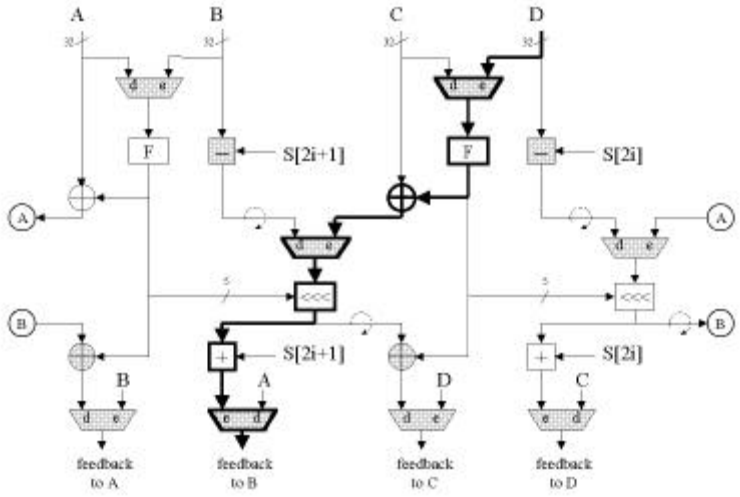


Fig. 11 Combinational part of a single round of RC6 implemented using basic architecture. Shaded components had to be added to the encryption unit, so it could perform decryption. The thick line shows the critical path in the circuit. Unit F performs operation $(2(X^2 \bmod 2^{32})+ X) \bmod 2^{32} <<< 5$. An arrow around a line means inverting the order of bits.

The relative size of the extra circuitry is the smallest for Mars and Twofish (less than 10%), and about 20% for RC6 (see Fig. 11). For Serpent and Rijndael, encryption and decryption are performed by two independent units of equal size. For Rijndael, these two units share 16 look-up tables implementing inversions in the Galois Field GF(2^8). These look-up tables take about 45% of the area used for encryption. Thus, the extra decryption circuitry takes for Serpent 100%, and for Rijndael about 55% of the area required for encryption itself.

Cipher	Extra logic	Extra logic area /encryption logic area
Twofish	2 32-bit XOR2, 2 32-bit MUX2	6%
Mars	2 SUB32, 3 32-bit MUX2	3%
RC6	2 SUB32, 2 32-bit XOR2, 8 32-bit MUX2 (see Fig. 11)	20%
Rijndael	Decryption independent of encryption, except 16 S-boxes 8x8	55%
Serpent	Decryption independent of encryption	100%

Table II. Extra logic that must be added to the circuit implementing encryption, so that the modified circuit can perform both encryption and decryption. Notation: XOR2 - 2-input XOR, MUX2 - 2-input multiplexer, SUB32 - 32-bit subtractor.

5.4 Critical path

The critical paths of all five AES candidates are characterized in Table III. As an example, the critical path of RC6 (without init MUX) is shown in Fig. 11.

Based on the characteristics of the critical path, the AES candidates can be divided into two main categories. Ciphers from the first category, RC6 and Mars, include in the critical path one complex arithmetic operation, such as modular multiplication or modular squaring, which determines the minimum clock period of these ciphers. The second category includes Rijndael, Twofish, and Serpent. In these ciphers, the critical path includes one or several S-boxes, and several multiple-input XORs. The minimum clock period is the sum of the access time to memories used to implement S-boxes, and delays introduced by multiple-input XORs and other simple auxiliary operations. The critical path of Twofish contains additionally two 32-bit additions.

The effect of resource sharing between encryption and decryption on the critical path is the strongest for RC6 (three encryption/decryption multiplexers in the critical path), very small for Rijndael, Twofish and Mars (one encryption/decryption multiplexer in the critical path), and negligible for Serpent. In Mars, additional delay (2 multiplexers) is caused by sharing resources between the forward and backwards keyed transformations.

Cipher	Minimum clock period - Virtex [ns]	Minimum clock period - XC4000 [ns]	Number of rounds	Components in the critical path (path flow / list of operations)
Rijndael	38.6	-	10	E/D MUX → S-box → affine transformation → MixColumn → init MUX
				S-box 8x8, XOR6, XOR5, XOR4, XOR2, 2 MUX2
Twofish	45.1	88.0	16	S-box → MDS → PHT → key addition → xor → E/D MUX → init MUX
				6 S-box 4x4, 2 ADD32, 9 XOR2, XOR4, XOR5, 2 MUX2
Serpent	94.3	-	4	8 x {key mixing → S-box → linear transformation) → init MUX
				8 S-box 4x4, 8 XOR2, 8 XOR7, MUX2
RC6	61.6	139.5	20	E/D MUX → squaring → addition → xor → E/D MUX → variable rotation → addition → E/D MUX → init MUX
				SQR32, 2 ADD32, ROT32, XOR2, 4 MUX2
Mars	100.6	-	32	2 mode MUXes → E/D MUX → multiplication → XOR → init MUX
				MUL32, XOR2, 4 MUX2

Table III. Critical paths in the implementation of the basic architecture for all AES candidates. Notation: E/D MUX - encryption/decryption multiplexer, i.e., multiplexer used to change the data flow between encryption and decryption; mode MUX - multiplexer used to change the data flow depending on the mode of transformation (e.g., forward and backwards transformation in Mars); init MUX - multiplexer used to select between loading a new block of data and feeding back data from the end of the cipher round (the only multiplexer shown in Fig. 3a); XOR*n* - *n*-input XOR, MUX2 - 2-input multiplexer, ADD32 - 32-bit adder, MUL32 - 32-bit multiplier mod 2^{32}, SQR32 - 32-bit squaring mod 2^{32}, ROT32 - variable rotation of a 32-bit word.

5.5 Area critical components

The components contributing most to the circuit area, for each AES candidate, are shown in Table IV. The ciphers fall clearly into two groups: Twofish and RC6 have the area approximately three to four times smaller than the area of the remaining three candidates, Mars, Rijndael, and Serpent. The relatively small area of Twofish and RC6 comes from the fact that both ciphers are of the Feistel type. The relatively large size of Serpent and Rijndael comes from the fact that both ciphers are S-P networks, and the amount of resource sharing between encryption and decryption is limited (no resource sharing for Serpent, about 45% resource sharing for Rijndael). Additional factor contributing to the large size of Serpent is the use of eight different types of S-boxes in eight subsequent cipher rounds.

Cipher	# of CLB slices - Virtex	# of CLBs - XC4000	Area critical components
Twofish	1076	907	96 S-box 4x4 (6 kbit), 18 32-bit XOR2, 24 MUL GF(2^8)
RC6	1139	1222	2 SQR32, 12 32-bit MUX2, 2 ROT32
Serpent	4438	-	512 S-box 4x4 (32 kbit), 2048 XORn (linear transformation, n=2..7)
Mars	2737	-	4 S-box 8x32 (32 kbit), MUL32, 22 32-bit MUX2
Rijndael	2902	-	16 S-box 8x8 (32 kbit), 24 MUL GF(2^8), 256 XOR5 (affine and inverse affine transformation)

Table IV. Cipher components contributing most to the circuit area. Notation: MUL GF(2^8) - multiplication in the Galois Field GF(2^8), XOR*n* - *n*-input XOR, MUX2 - 2-input multiplexer, MUL32 - 32-bit multiplier mod 2^{32}, SQR32 - 32-bit squaring mod 2^{32}, ROT32 - variable rotation of a 32-bit word.

The relatively large size of Mars is the result of the design decisions, such as

a. using two different kinds of rounds (mixing vs. keyed transformation). For the basic non-pipelined architecture, only one type of round is active at a time.
b. using 4 large S-boxes 8x32 in a single round of the mixing transformation. Sharing two of these S-boxes during mixing transformation is possible only at the cost of doubling the number of clock cycles required for this transformation. (Our implementation still shares two S-boxes between the mixing transformation and the keyed transformation.)
c. using area-consuming 32x32 bit modular multiplication.

The area of Mars, Serpent, and Rijndael is dominated by S-boxes. Even though the number and size of these S-boxes is very different for each cipher, the total number of bits in memories implementing S-boxes, 32 kbits, is identical for all three ciphers. This may explain the relatively similar size of all three implementations expressed in number of CLBs.

5.6 Potential for inner-round pipelinig

Inner round pipelining can be most effectively applied to the ciphers with the following features:

a. the cipher round is composed of a large number of layers, with all layers performing simple operations with comparable delays;
b. the cipher round does not contain large hard-to-divide functional units.

Additionally, for FPGA implementations, it is advantageous if the implementation of the basic architecture contains large number of CLBs with unused flip-flops (one bit registers).

The above conditions are the best fulfilled by Serpent. It is straightforward to introduce 8 internal pipeline stages to the implementation round of Serpent (one implementation round = 8 regular cipher rounds), one after each regular cipher round. Implementing pipeline stages inside of the regular cipher round is possible in theory, but may be difficult in practice because of the clock frequency limitations imposed by the control unit.

The second cipher best suited for inner-round pipelining is Twofish. According to Table III, the critical path of Twofish contains a large number of simple operations with comparable delays, including a 4x4 S-box read-out, XOR operations, and additions. The most complex of these operations is a 32-bit addition. It is likely that this operation may need to be implemented using multilevel carry-lookahead architecture to take the full advantage of the inner-round pipelining in Twofish. Additionally, the FPGA implementation of basic architecture of Twofish contains a relatively small number of unused flip-flops, which will cause that the circuit area will increase by a larger percentage than for Serpent with the same number of inner-round pipeline stages.

Rijndeal is relatively easy to pipeline, but its critical path contains only 7 elementary operations. Additionally, the most time-consuming of these operations, the 8x8 S-box read-out, is hard to divide into extra pipeline stages. RC6 can be efficiently pipelined at the cost of increase in the circuit area resulting from using fast architectures for addition and multiplication (e.g., carry lookahead and carry save). Mars is the most difficult to pipeline because of the

a. irregular structure with different operations in various paths;
b. two types of rounds (mixing and keyed transformation) both using large S boxes;
c. need for the complex fast architectures for the pipelined multiplication and addition.

5.7 Potential for loop unrolling

The largest gain from loop unrolling can be achieved by ciphers with the following properties:

* small area used by the combinational part of a single round, which permits fitting a large amount of rounds in the largest available FPGA device;
* small delay of a single round compared to the sum of delays eliminated by loop unrolling, including the round multiplexer delay, the register delay, and the register setup time (as shown in formula (2)).
* potential for optimizations at the boundary between the last and the first operation of the cipher round.

Assuming the use of the largest available Virtex chip, RC6 and Twofish have the highest potential for loop unrolling. The largest Virtex chip can easily fit ten RC6 rounds and eight Twofish rounds. Mars can be implemented with four rounds unrolled; Rijndael and Serpent with only two rounds unrolled.

5.8 Potential for outer-round pipelining and mixed outer-inner-round pipelining

The largest gain from outer-round pipelining can be achieved by ciphers with the smallest area. The largest number of pipelined rounds fitting within the largest available Virtex chip is the same as in the architecture with loop unrolling. As a result, Twofish and RC6 can benefit most from the outer-round pipelined architecture. The throughput of both these ciphers exceeds 1 Gbit/s for the architectures with the maximum number of outer-round pipeline stages. Additional speed-up can be obtained by combining outer and inner round pipelining, leading to the mulitigigabit-per-second performance. For Serpent, the most straightforward form of mixed pipelining, with 16 regular cipher rounds unrolled and a register after each regular cipher round (1/8 of the implementation round), would result in an even higher performance. Mars can benefit substantially from both forms of pipelining; Rijndael primarily from the inner-round pipelining.

6. Design procedure and tools

The design flow and tools used in our group for implementation of algorithms in FPGA devices are shown in Fig. 12. All five AES ciphers were first described in VHDL, and their description verified using the functional simulator from Aldec, Inc. Test vectors and intermediate results from the reference software implementations were used for debugging and verification of VHDL codes. The revised VHDL code became an input to Xilinx tools performing the automated logic synthesis, mapping, placing, and routing. These tools generated reports describing the area and speed of implementations, a netlist used for timing simulations, and a bitstream to be used to program an actual FPGA device. A final step is to verify the design experimentally, using physical FPGA devices. We plan to perform these experiments using a PCI FPGA board from Virtual Computer Corporation [VCC]. The most complex PCI board currently available from VCC is based on the XC4062XL FPGA device. This device is able to fit full implementations of Twofish and RC6, and an encryption portion of Serpent. All details of our implementations and experiments will be described in the technical report [CG00].

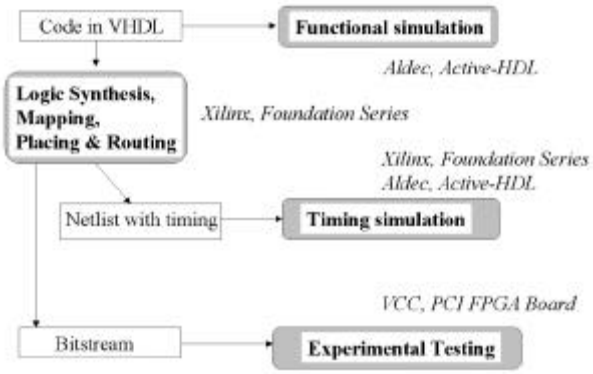


Fig. 12 Design flow for implementing AES candidates using Xilinx FPGA devices.

7. Need for interleaved operating modes

The full potential of hardware implementations of symmetric block ciphers can only be utilized in cipher modes that support efficient use of pipelining, as shown in Fig. 8. To date, the ECB mode is the only operating mode standardized by NIST that supports efficient pipelining. However, ECB is not regarded secure for transmissions of large volumes of data, and most standard protocols recommend using CBC or CFB modes instead. Therefore, we propose to speed-up the standardization effort, and include in the AES standard interleaved modes of operation, such as the interleaved CBC mode defined by:

$$C_i = \mathrm{AES}(M_i \oplus IV_i) \text{ for } i=1 \text{ to } N, \text{ and } C_i = \mathrm{AES}(M_i \oplus C_{i-N}) \text{ for } i>N\,. \quad (4)$$

The standard should support arbitrary values of the interleaving factor N, smaller than a certain maximum.

8. Conclusions

The results and analyses presented in this paper show that the differences in hardware performance of the AES candidates are bigger and more significant than the corresponding differences in software performance. No correlation between software and hardware performance was found. On the contrary, Serpent, believed to be the slowest candidate in software, appeared to be the fastest of the five AES candidates in hardware. We believe that the large differences among parameters of all five AES algorithms in hardware resulted primarily from internal structure of these algorithms, and were not significantly affected by our implementation decisions. On the other

hand, we could not completely eliminate or predict the influence of the FPGA design tools and the VHDL design entry method on the results of the comparison. Assessed exclusively from the hardware performance point of view, the five AES finalists fall into the three distinct classes with different performance characteristics.

The first class includes Twofish and RC6. Both ciphers guarantee compact low-cost implementations with medium speed compared to other candidates. In particular, because of the area constraints, Twofish and RC6 are the only ciphers that can be implemented using low cost FPGA devices from the Xilinx XC4000 family. Both ciphers can be substantially sped-up by outer-round pipelining (for non-feedback modes (ECB, counter mode)), and - to the lesser extent - by loop-unrolling (for cipher feedback modes (CBC, CFB)). Among the two, Twofish is in some respects superior to RC6. It is about 70% faster and is more suitable for inner-round pipelining. Both ciphers use comparable area, and as a result their potential for loop unrolling and outer-round pipelining is similar.

The second class includes Serpent and Rijndael. Both ciphers guarantee very high speed at the cost of the relatively large area compared to the ciphers from the first class. The primary way of speeding up these ciphers for non-feedback cipher modes (ECB and counter mode) is inner-round pipelining. Both ciphers have a similar speed in the basic architecture. Rijndael can be implemented using about 35% less area. The more regular architecture of Serpent makes it significantly more suitable for a multi-stage inner-round pipelining.

The third class is composed of Mars itself. This cipher shows the worst hardware characteristics of all five candidates. It is over twice as slow than the next slowest candidate (RC6), and over 8 times slower than the fastest AES cipher (Serpent). It also takes over twice the area used by ciphers from the first group, Twofish and RC6. Further optimizations of the Mars implementation are certainly possible, but would require the higher development effort than that devoted to other AES candidates.

It is interesting to notice that although four out of five candidates outperform Triple DES in terms of speed, only Twofish has a comparable performance in terms of the speed/area ratio. Three other candidates, Rijndael, RC6, and Serpent, have a similar, and much lower than triple DES, value of this performance parameter.

Out of all five candidates, Twofish seems to be the most suitable for applications where the primary requirement is the limited cost or area of the cryptographic hardware. Serpent and Rijndael both offer superior performance for applications where the speed itself is a criterion of primary concern.

Acknowledgments

The authors would like to thank Christof Paar and his students, as well as Miles Smid and other members of the NIST Computer Security Division for valuable comments and discussions. We also would like to thank our students, Po Khuon and Tanvir Joy, for their work on implementation of Triple DES.

Literature:

[BCD+98] C. Burwick, D. Coppersmith, E. D'Avignon, R. Gennaro, S. Halevi, C. Jutla, S. M. Matyas, L. O'Connor, M. Peyravian, D. Safford, and N. Zunic, "Mars - A Candidate Cipher for AES," NIST AES Proposal, June 1998.

[CG99] P. Chodowiec and K. Gaj, "Implementation of the Twofish Cipher Using FPGA Devices", Technical Report, George Mason University, July 1999; available at http://www.counterpane.com/twofish.html.

[CG00] P. Chodowiec and K. Gaj, "Implementations of the AES Candidate Algorithms using FPGA Devices," Technical Report, George Mason University, April 2000 (to be published on the web).

[EP99] A.J. Elbirt and C. Paar, "An FPGA Implementation and Performance Evaluation of the Serpent Block Cipher," Eighth ACM International Symposium on Field-Programmable Gate Arrays, Monterey, California, February 10-11, 2000. Preprint available at http://ece.wpi.edu/Research/crypt/publications/index.html.

[NBD+99] James Nechvatal, Elaine Barker, Donna Dodson, Morris Dworkin, James Foti, Edward Roback, "Status Report on the First Round of the Development of the Advanced Encryption Standard," NIST report, August 1999.

[NSA98] National Security Agency, "Initial plans for estimating the hardware performance of AES submissions," http://csrc.nist.gov/encryption/aes/round2/round2.htm.

[RH99] M. Riaz and H. Heys, "The FPGA Implementation of RC6 and CAST-256 Encryption Algorithms," accepted for CCECE'99, Edmonton, Alberta, Canada, 1999.

[RRS+98] R. Rivest, M. Robshaw, R. Sidney, and Y. L. Yin, "The RC6 Block Cipher," NIST AES Proposal, June 1998.

[SKW+98] B. Schneier, J. Kelsey, D. Whiting, D. Wagner, C. Hall, N. Ferguson, "Twofish: A 128-Bit Block Cipher," NIST AES Proposal, June 1998.

[SKW+99] B. Schneier, J. Kelsey, D. Whiting, D. Wagner, C. Hall, N. Ferguson, "Performance Comparison of the AES Submissions," Second AES Candidate Conference, Rome, April 1999.

[VCC] Virtual Computer Corporation, http://www.vcc.com/

Session 2:

"Platform-Specific Evaluations"

AES Finalists on PA-RISC and IA-64: Implementations & Performance

John Worley, Bill Worley, Tom Christian, Christopher Worley[1]
Hewlett Packard Labs
Fort Collins, CO

Overview

The Advanced Encryption Standard selection process has, for the first time, included software execution speed as a relevant criterion for the choice of the next standard. The initial submissions included keying, encryption, and decryption execution times, in clock cycles, for Intel Pentium, Pentium II, and Pentium Pro microprocessors. While Pentium execution speeds are important, by no means do they completely characterize software performance, particularly that of existing RISC microprocessors and the new IA-64 microprocessor family.

In order to enable a more complete characterization of software performance, our group, working from HP Labs, decided in January 1999, to study and publish the performance of likely AES finalists for PA-RISC and IA-64 microprocessors. We initially selected RC6, Rijndael, Serpent, and Twofish. Our preliminary results were informally presented at the 1999 Rome Conference. Following the selection of the five finalists, we included work on MARS. This paper discusses the issues, implementations, and results of our work for each of the five AES finalists.

Details of specific engineering tradeoffs for Itanium and McKinley chips remain proprietary. We therefore are not at liberty to disclose complete source codes and performance details from which such information can be deduced. What we have chosen to present are actual simulation cycle counts for a snapshot of the evolving McKinley design. These are not cycle counts for an actual product. We offer them as well-substantiated, conservative indicators of the performance of the future family of IA-64 processors. Itanium will be somewhat slower; future implementations will be faster. We believe these results do provide a reasonable basis for software performance judgments about the AES finalists. A summary table appears at the end of the paper.

In addition to processor cycle count, we also present PA-RISC and IA-64 code sizes, register usage, and instruction-level parallelism. Finally, we describe the programming approaches we employed for effective use of both architectures. We would be happy to share full details with the finalists' authors under non-disclosure terms.

Methodology

We focused on hand-optimized assembly language implementations of the algorithms for 128-bit keys and 128-bit blocks, using compiled codes as sanity checkers. We agree with Bruce Schneier that AES codes will be implemented in this manner in actual systems; this also leads to the clearest comparisons between instruction set architectures. Codes for this study were optimized for performance, not code size or table size.

For PA-RISC we measured execution speeds on a PA-8500. We timed executions using the PA-RISC 64-bit interval timer, which counts actual clock cycles. To eliminate cache and system effects, we ran tens of millions of executions, varying keys and data blocks on a lightly loaded system, and profiled those runs with minimum cycle counts. We observed that runs often would differ by only a few cycles, and that the cycle counts formed Gaussian distributions. It was further observed that the input value (input key for keying, data block for encryption/decryption) noticeably affected performance for algorithms that used table look-ups. Thus, while the PA-RISC times are best observed times, we also show the distribution's average and maximum values.

Lacking IA-64 hardware, we employed three different types of simulators. Initial debugging used a fairly fast and purely functional instruction set simulator. The second type was considerably slower, but simulated parallel execution, latencies, and memory hierarchy behavior. This was used for additional code validation and preliminary execution cycle counts.

These simulators, while useful, did not guarantee absolute fidelity to the chip designs. Therefore, final timings used fully simulated RTL designs of the Merced (now Itanium) and McKinley chips. This approach was extremely slow, and our results often varied from day to day, as engineers improved their designs. We constructed special tools that automatically prepared test inputs and displayed the cycle-by-cycle behavior of the microprocessor pipeline. The memory hierarchy was initialized for each run, and the timing could be computed by subtracting cycle numbers from the pipeline output.

Notation

`A <<< n`	Left rotation by `n` bits
`A >>> n`	Right rotation by `n` bits
`A ⊕ B`	Bit-wise Exclusive-OR
`A +.× B`	Matrix multiplication
`[b0, b1, ..., bn]`	Column vector, LSB first

PA-RISC Facts

PA-RISC first shipped in 1986 and is the processor for Hewlett-Packard's RISC workstation and server products. Architecture features include 64-bit virtual addressing, 32 general-purpose registers, and 32 floating point registers. Current processors implement the 64-bit Version 2.0 of the PA-RISC architecture.

[1] John S. Worley **jworley@fc.hp.com** William S. Worley, Jr. **worley@hpl.hp.com**
Tom W. Christian **twc@fc.hp.com** Christopher S. Worley **cworley@fc.hp.com**

This study utilized the PA-8200 and PA-8500 microprocessor chips. Both of these chips are out-of-order superscalar designs, capable of executing two memory operations and two integer or floating point instructions per cycle. Only one store instruction can complete per cycle. Careful software scheduling is required to realize the full parallelism.

IA-64 Overview

This section provides a *very* brief overview of the IA-64, highlighting features in the discussions that follow. Readers familiar with the architecture can skip this section.

Parallelism and Functional Units

The majority of processor architectures specify sequential instruction execution. Microarchitectures then employ superscalar logic to issue multiple instructions in parallel whenever possible. In contrast, the IA-64 architecture puts all the parallelism cards on the table. There are four types of functional units: **M** (memory), **I** (integer), **F** (floating point), and **B** (branch); each IA-64 implementation has two or more of each of these units. IA-64 hardware detects when program parallelism exceeds the capabilities of the implementation, but responsibility for organizing instructions to execute in parallel is wholly with the programmer or compiler.

Instructions, Bundles, and Issue Groups

There is a corresponding instruction class for each functional unit type, although a specific instruction may not be able to execute on all units of that type in a given implementation. In addition, there is an **A** (ALU) instruction class that can execute on both **I** and **M** units. **A** instructions include most integer arithmetic and logical operations, so that otherwise idle memory units can be used for parallel computation.

Three instructions are grouped into a *bundle*, where all instructions in the bundle may be eligible to be issued in parallel to functional units specified by the bundle type. Sequential bundles that can issue in parallel form an *issue group*. One characteristic of an IA-64 implementation is the maximum number of bundles that can issue together. For example, a processor that can issue at most two bundles in one cycle is referred to as a "two-banger."[2]

Registers and the Register Stack

IA-64 provides 128 64-bit integer registers. The low 32 registers (`r0` - `r31`) are common for all code. For function arguments and local values, each procedure can allocate up to 96 additional registers in a *register stack frame*. Saving and restoring registers in the register stack is handled by an independent hardware thread, so that no registers need to be saved and restored explicitly.

In addition to the integer registers, IA-64 provides 128 extended precision (64-bit mantissa, 17-bit exponent) floating point registers, 64 1-bit predicate registers (see below), and eight branch registers for indirect branches.

Predication

A powerful feature of IA-64 is *instruction predication*. Every instruction, except for certain branch and control instructions, is predicated, i.e., its execution is enabled or disabled by one of the 64 predicate bits. One predicate, `p0`, is hardwired to '1' for instructions that execute unconditionally or cannot be predicated. Predicates are set or cleared by compare instructions and certain floating-point instructions. Also, the 64 predicates can be read or set in parallel using special instructions. Predication allows, for example, one of two instructions to execute based on a comparison condition, or for instructions to be enabled during the first pass of a loop and disabled for all subsequent iterations.

Counted Loops

IA-64 provides hardware support for counted loops. The special registers `ar.lc` (loop counter) and `ar.ec` (epilogue counter) control when the branch instructions `br.ctop` and `br.cexit` are taken. For example, if `ar.lc` is set to 9 and `ar.ec` is set to 0, a counted loop will execute 10 times if the loop ends with `br.ctop`, 9 times if the loop begins with `br.cexit`. The hardware is designed to predict perfectly when a branch will be taken or fall through, so that counted loops can execute with no branch penalties.

Rotating Registers

When a subroutine allocates a register stack frame, some or all of the local registers, starting from `r32`, can be set to *rotate*. Each time a counted loop branch is taken, the rotating registers are circularly renamed such that the next iteration of the loop can operate on different data without changing the register name. For example, if there are eight registers designated as rotating, the renaming is as follows:

`r32` → `r33` → `r34` → `r35` → `r36` → `r37` → `r38` → `r39` → `r32`

Fixed portions of the floating point and predicate registers also rotate. The high 96 floating point registers (`f32` through `f127`) rotate. The high 48 predicate registers (`p16` to `p63`) also rotate, but with a slight difference. While the loop counter `ar.lc` is non-zero, a '1' value is shifted into `p16`; if `ar.lc` is zero and the epilogue counter `ar.ec` > 1, a '0' value is shifted in instead.

Programming Issues

There are three operations commonly used in cryptographic algorithms that are not fully realized in the integer hardware on PA-RISC and IA-64: fixed 32-bit rotations, variable 32-bit rotations, and 32x32→32 unsigned integer multiplies.

[2] This term comes from the slang term for a two-cylinder engine. While three-banger or more implementations are foreseeable, it seems unlikely that IA-64 will ever give rise to, say, a V12.

PA-RISC

On PA-RISC, fixed rotations can be executed in one cycle using the shift right pair word (`shrpw`) instruction. This instruction concatenates the low 31 and 32 bits from left and right source registers, respectively, shifts right the specified distance, and leaves the high 32 bits undefined. If the two source registers are the same, the low bits are concatenated with the high bits, exactly as would occur in a rotation. Thus, fixed rotations on PA-RISC can be defined as follows:

```
ROTR   .macro       src, dst, count
       shrpw        src, src, count, dst
       .endm
ROTL   .macro       src, dst, count
       shrpw        src, src, 32 - count, dst
       .endm
```

Variable rotations use the same strategy, except that an extra cycle is required to move the shift distance into the SAR (shift amount register). For a right rotation, the actual shift distance is used. For a left rotation, the 5-bit complement of the distance is used and the value is pair-shifted right one before the variable shift. The left shift also executes in two cycles since the `mtsarcm` (move to SAR complement) and the first `shrpw` can issue in the same cycle on the PA-8000 family.

Integer multiplication on PA-RISC requires using the unsigned integer multiply in the floating point unit. Since the only path for moving data between the integer and floating point units is memory, the multiplicands must be stored, loaded into the FPU, multiplied, stored again, and reloaded into the integer unit. This adds latencies on both sides of the multiply, in addition to the multiply time itself.

IA-64

Although the IA-64 architecture has a shift right register pair instruction, it only operates on full 64-bit registers. This can still be used to implement 32-bit fixed rotations in two cycles as follows:

```
dep.z  TMP = src, 32, 32
shrp   dst = src, TMP, count + 32
```

for right rotations, and

```
dep.z  TMP = src, 32, 32
shrp   dst = src, TMP, 64 - count
```

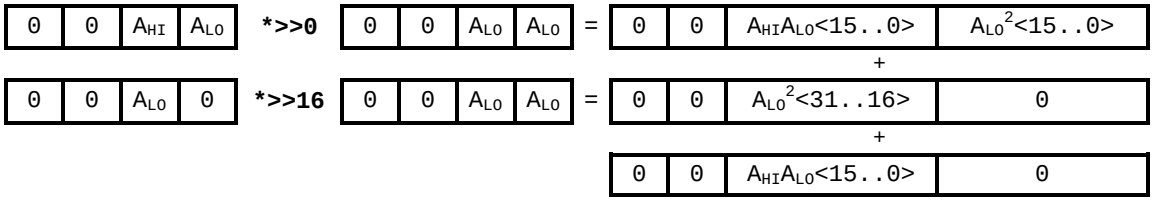
for left rotations.

The `dep.z` instruction puts the low 32 bits of the source register in the high half of a temporary register, clearing the low half. The pair-shift concatenates the low bits with the high bits and shifts far enough to put the proper set of bits in the low half of the destination. Like the PA-RISC instruction, the destination's high half is not cleared. None of the AES finalists require these bits to be cleared; however, the `zxt4` instruction can be used if necessary.

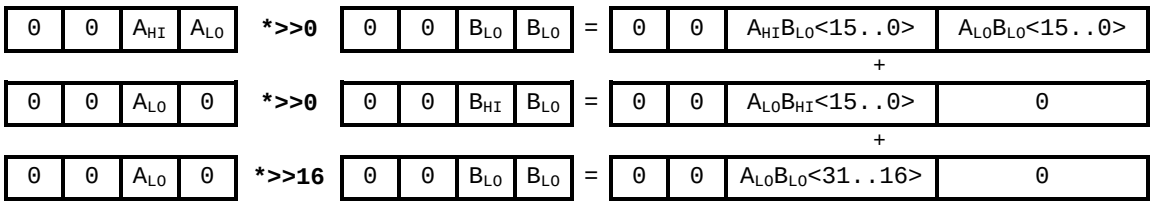
On IA-64, variable rotates are implemented much as in the C language: shift left `j`, shift right `(32 - j)`, `OR` or `ADD` the results together. This involves four operations and a minimum of three cycles. The variable shifts are executed on the multimedia units (MMUs).

Like PA-RISC, the IA-64 primary integer multiply is implemented on the floating point unit and involves latency cycles to move back and forth. However, 16x16 MMU multiplies and parallel adds can be used to compute and sum the partial products instead. This is effective when only the low 32 bits of the result are of interest. In particular, the parallel 16-bit unsigned multiply and shift instruction (`pmpyshr.u`) can be used to complete a 32x32→32 multiply.

If we consider multiplicands derived from `A` as four 16-bit elements, A^2 can be computed with two multiplies and two adds as follows:



One of the operands is just the argument, `A`. The other two arguments are generated by the 16-bit `mux` MMU instructions; the additional addend is derived from the first product using the 16-bit `mix` instruction. The general 32x32→32 requires three multiplies and two additions. If we consider multiplicands derived from `A` and `B` as four 16-bit elements, the operations are:



Two of the operands are just the arguments, `A` and `B`. The other two arguments are generated by the 16-bit `mix` and `mux` MMU instructions.

It has been noted that with better hardware support for 32-bit rotations and 32x32→32 multiplication, all the AES finalists will outperform Pentium on IA-64. In the performance analysis for each algorithm, we have estimated performance for a hypothetical IA-64 implementation, called IA-64++, with the following enhancements:

- A single-cycle shift right pair word instruction, as in PA-RISC
- Single-cycle, 32-bit, left and right variable rotate instructions
- A two-cycle 32x32→32 unsigned multiply

Mars

The Mars encryption scheme (IBM team) uses a mix of approaches: substitution boxes, Feistel networks, multiplication, and fixed and variable rotates. The single substitution box, `S[]`, is fixed, and is employed both as a 512 word array (9-bit index), and as low (`S0[]`) and high (`S1[]`) 256 word arrays (8-bit index). The principal challenges for PA-RISC and IA-64 implementations are the 32x32→32 multiply and variable rotates.

Keying[3]

Mars keying initializes the first `N` elements of a fifteen-element array, `T[]`, to the input key `k[]`, where N is the size of the key in 32-bit words. The key is then padded to 15 words by setting `T[N]`←`N` and zeroing the remainder of the array. Instead of generating the entire expanded key directly, Mars generates ¼ of the array, or 10 words, each time, repeating the process four times to develop the entire key array, `K[]`. There are three steps in each iteration: linear transform, stirring, and storing. The linear transform applies the formula:

```
T[i] = T[i] ⊕ ((T[i-7 mod 15] ⊕ T[i-2 mod 15]) <<< 3) ⊕ (4i + R)
```

to each element of the array, where `R` is the iteration count (0..3). Stirring uses the following formula:

```
T[i] = (T[i] + S[T[i-1 mod 15] & 0x1ff]]) <<< 9
```

applied to each word, repeated four times. Finally, 10 words from the intermediate array are stored in the expanded key array as follows:

```
K[10×R + i] = T[4i mod 15]
```

which effectively stores words 0, 4, 8, 12, 1, 5, 9, 13, 2, and 6, in that order, from the temporary array. After all the expanded key words are generated, those used in multiplication (`K[5]`, `K[7]`, …, `K[35]`) are modified if they are weak, i.e., contain long runs of 1's or 0's. The algorithm for identifying weak key words comes from the Mars implementation by Brian Gladman.

PA-RISC

The PA-RISC implementation keeps `T[]` in registers. The linear transform, the inner stirring loop, and key stores are straight-lined. In the fix-up phase, the two-ALU PA-RISC has sufficient execution bandwidth to compute the fix-up mask in parallel with looking for long runs of 1's or 0's. If there are no such runs, the remainder of the fix-up is skipped. Using the authors' estimates that statistically 1 out of 41 keys are weak, the extra computation is skipped 97.6% of the time, a performance win even with a branch penalty.

IA-64

The IA-64 Mars keying implementation uses software pipelining to increase keying speed. The routine allocates a 16-register stack frame, all of which are rotating. The register usage is as follows (indices are modulo 15):

r32	r33	r34	r35	r36	r37	r38	r39	r40	r41	r42	r43	r44	r45	r45	r47
T_{i-1}	T_{i-2}	T_{i-3}	T_{i-4}	T_{i-5}	T_{i-6}	T_{i-7}	T_{i-8}	T_{i-9}	T_{i-10}	T_{i-11}	T_{i-12}	T_{i-13}	T_{i-14}	T_i	T_x

By assigning $T_x \leftarrow T_i$ at the end of the loop, this organization implements a 15-register rotation. The linear transform XORs T_{i-2} (`r33`) and T_{i-7} (`r38`), rotates the result, the XORs with T_i and the iteration constant (`4i + R`). This would normally require four cycles; however, the transform can be reorganized into a two-stage, two-cycle pipeline. The first stage computes $T_{i-2} \oplus T_{i-7}$ and extracts the high three bits of the result; the second phase computes $T_i \oplus$ (`4i + R`) and completes the rotation, then XORs the final result. The loop uses rotating predicates to disable the second phase on the first iteration, while the last execution of the second phase is handled after the loop so that the values return to their initial positions when the loop is complete.

Pipelining the inner stirring loop is limited by the use of `T[i-1 mod 15]` in computing `T[i]`; however, the high-order nine bits extracted for rotation can be used to start the S-Box look-up for the next iteration. This allows a two-stage, four-cycle pipeline, which executes 33% faster than the 6-cycle, non-pipelined equivalent.

Like PA-RISC, the fix-up mask can be computed in parallel with looking for runs of 1's and 0's. Unlike PA-RISC, branches include 'hints', so that the branch penalty is only incurred for weak keys, or 2.4% of the time.

Encryption

Mars encryption consists of four phases, each repeated eight times: forward mix, forward keyed transform, backward keyed transform, and backward mix. The forward and backward mixing uses table look-ups, fixed rotation, XORs, and addition and subtraction in a rotating pattern, e.g., `fmix(A, B, C, D)`, `fmix(B, C, D, A)`, etc. There are asymmetric additions in steps 1, 2, 4, and 5 of the forward mix, with corresponding subtractions in steps 2, 3, 5 and 6 of the backward mix.

[3] This is the 'tweaked' version of the Mars keying. The implementation of the initialization, mixing, and stirring phases of the original scheme is discussed in Appendix B. The key fix-up is identical for both schemes.

The core of the keyed transforms is the *E* function, which takes one data word and uses table look-up, multiplication, variable rotation, additions and XORs to generate three data words (`L`, `M` and `R`) to add or XOR with the other three data words as follows:

Forward Mode	Backward Mode
`D[1] += L`	`D[1] ^= R`
`D[2] += M`	`D[2] += M`
`D[3] ^= R`	`D[3] += L`

On PA-RISC, the mixing phases are coded as straight-line operations. Even with the four table look-ups per step, there is enough memory bandwidth to load the 16 multiplicative keys into the floating-point unit at the same time. The real bottleneck is the integer multiply in the *E* function: the data word must be rotated, stored, loaded into the floating point unit, multiplied, stored again and reloaded into an integer register. Although an addition and table look-up can be evaluated in parallel, these do not fully amortize the performance cost of the multiply.

On IA-64, both forward and backward mixing can be coded as a single loop: the asymmetric operations are controlled by loading a specific bit pattern in the rotating predicates, enabling the appropriate operation at the proper step. Because of perfect branch prediction with counted loops, this approach executes in the same cycle count as straight-line code.

On IA-64, MMU multiplies are used to compute the *E* function multiplication. Once the multiplication is complete, the remainder of the *E* function can be evaluated. Like the mixing phases, a predetermined bit pattern loaded in the rotating predicates controls whether the forward or backward mode operations are enabled at each step.

Performance

Cycles	Pentium	PA-RISC			IA-64	IA-64++
		Min	Average	Max		
Keying	2128	1797	1804.65	1879	1408	1408
Keying (Original)	3894	1969	1975.89	2060	1903	1313
Encryption	320	540	563.01	584	511	255
Decryption	374	538	552.37	566	527	271

On PA-RISC, Mars keying executes in 1797 cycles, compared to the best-reported Pentium results of 2128, a 15.6% performance advantage. Encryption and decryption, however, run slower due to the multiplication overhead: 68.8% slower for encryption (540 vs. 320) and 43.9% slower for decryption (538 vs. 374).

On IA-64, keying completes in 1408 cycles, a 33.8% performance gain. Encryption and decryption, with the extra cycles required for multiplication and variable rotation, are slower than Pentium: 59.7% slower for encryption (511 vs. 320) and 40.9% slower for decryption (527 vs. 374). Keying on IA-64++ is the same 1408 because the software pipelines hide the extra cycles needed for rotation. Encryption improves to 255 cycles (20.3% faster than Pentium), and decryption also improves to 271 cycles (27.5% faster).

RC6

The principal programming challenge when implementing RC6 (Rivest, Robshaw, Sidney, Yin) on PA-RISC and IA-64 is the lack of the fast 32x32→32 multiply and variable rotate primitive the algorithm requires for performance. On the positive side, IA-64's rotating integer registers and instruction predication simplify data management and allow for a very compact code size.

Keying

RC6 keying starts with the input key, `L[]`.The key array, `S[]`, is initialized using the two magic numbers P_{32} = 0xB7E15163 and Q_{32} = 0x9E3779B9, as follows:

```
S[0] = P32
S[1] = P32 + Q32
S[2] = P32 + 2 * Q32
S[3] = P32 + 3 * Q32
       . . .
```

The keying algorithm then performs three mixing passes over the two arrays:

```
A = S[i] = (S[i] + A + B) <<< 3
B = L[j] = (L[j] + A + B) <<< (A + B)
```

where A and B are initially zero, and i and j count circularly through the key and input key arrays, respectively. If the first pass through the key array is handled separately, it is possible to combine the key array initialization with the first mixing phase. The first mix can also be partially hard coded, since A = B = 0, and S[0] = P_{32}. Since, after the first loop pass, B is just the previous, modified input key word, the variable B is replaced with LPREV(k), the user input key L[(k-1) mod 4]. The first pass is coded as follows:

```
keyVal = P32;
A = T = ROTL(P32, 3);
for (k = 1; k < NKEYS; ++k) {
       LPREV(k) = ROTL(LPREV(k) + T, T);
       keyVal += Q32;
       S[k - 1] = A;
       A = ROTL(keyVal + A + LPREV(k), 3);
       T = LPREV(k) + A;
}
S[NKEYS - 1] = A;
```

This organization saves one full load and store of the key array and does not require computing the modulus 2*r + 4, where r is the number of encryption rounds. The last two passes are identical, with a similar structure to the first pass, but do not, of course, re-initialize the key array.

For PA-RISC, each instance of the loop can be unrolled four ways, with the input key words reordered circularly each time - this eliminates loading and storing the keys, and the modulus computation on the input key index.

The IA-64 architecture suggests a different strategy for implementation. The large register file allows *the entire key array* to be kept in registers; the rotating integer registers naturally mimic the way data flows through the computation, such that no indexing or modulo operations are required. The keying routine allocates a 56-register stack frame, all of which are rotating. The rotating registers are allocated as follows:

r32-r33	r34	r35	r36	r37	r38	r39	r40	r41	r42-r82	r83	r84-r87
Unused	L_X	L_n	L_{n+1}	L_{n+2}	L_{n+3}	S_X	S_{Active}	S_{Prev}	Key Array	S_{Next}	Unused

where <L_n … L_{n+3}> are initialized from the user input key. In order to circulate the keys and key array separately, $L_X \leftarrow L_{n+3}$ and $S_X \leftarrow S_{Next}$ before the registers are rotated. Each time through the loop, the code operates on L_n, S_{Active}, and S_{Prev}. Rewriting the mixing loop in these terms:

```
for (k = 1; k < NKEYS; ++k) {
       Ln = ROTL(Ln + T, T);
       A = ROTL(SActive + SPrev + Ln, 3);
       T = Ln + A;
       SActive = A;
       LX = Ln+3;
       SX = SNext;
}
```

Predicated instructions enable key array initialization during the first mixing pass and storing the final key words during the final pass, all within the same code loop and *without* branching. There are enough unused instruction slots to compute the two qualifying predicates with no additional cycles. The keying routine is thus coded in a single loop:

```
for (k = 1; k < 3 * NKEYS; ++k) {
       Ln = ROTL(Ln + T, T);       if (firstMix) SActive = SPrev + Q32;
       firstMix = k < NKEYS-1;     lastMix = k >= 2 * NKEYS;
       SPrev = A;                  if (lastMix) *S++ = A;
       A = ROTL(SActive + A + Ln, 3);
       T = Ln + A;
       LX = Ln+3;
       SX = SNext;
}
*S = A;
```

This coding is extremely compact: the entire routine consists of 39 instructions in 16 IA-64 bundles; the core loop is 20 instructions.

Encryption

The RC6 definition is compact and elegant, but the algorithm relies on a fast 32x32→32 multiply and variable rotate for performance. To multiply on PA-RISC, the two data words must be stored, loaded into the floating point unit, multiplied, stored again and reloaded into integer registers. The inner loop is unrolled to rotate the data words.

On IA-64, MMU multiplies are used to compute A^2. Once the full multiplication is complete, the `shladd` instruction computes the final product $2A^2 + A \equiv A*(2A + 1)$. Using rotating registers for the data words, RC6 encryption can be coded in a single loop.

Performance

Cycles	Pentium	PA-RISC			IA-64	IA-64++
		Min	Average	Max		
Keying	1632	1077	1077	1077	1581	1057
Encryption	243	580	590.76	597	490	150
Decryption	226	493	496.37	499	490	130

On PA-RISC, RC6 keying executes in 1077 cycles, compared to the best-reported Pentium results of 1632, a 34% performance advantage. Encryption and decryption, however, run slower due to the multiplication overhead: 138% slower for encryption (580 vs. 243) and 118% slower for decryption (493 vs. 226).

On IA-64, keying completes in 1581 cycles, a 3.1% performance gain. Encryption and decryption, with the extra cycles required for multiplication and variable rotation, are slower than Pentium: 101.7% slower for encryption (490 vs. 243) and 116.8% slower for decryption (490 vs. 226). For IA-64++, keying is estimated to run in 1057 cycles, 54% faster than Pentium, encryption in 150 cycles (38.3% faster), and decryption in 130 cycles (42.5% faster)

Rijndael

The principles for a fast Rijndael (Daemen, Rijmen) implementation are largely explained in the algorithm specification. A short comment in section 5.2.2 summarizes the general approach:

> *"In the table-lookup implementation, all table lookups can in principle be done in parallel. The EXORs can be done in parallel for the most part also."*

This turns out to be an understatement. In other AES candidates, parallelism must be squeezed from the specification, while Rijndael's parallelism cup runneth over. Even the keying phase has considerable parallelism, as will be shown.

Realizing this parallelism requires five 4K tables, as discussed below, although only two tables are used for any one operation. Each 4K table is made up of 4 256x4 byte tables, where each 1K table is rotated one byte position from the previous. The tables and the operations they're used in are:

S-Box	**Keying, Encryption** Implements byte substitution only
I-Box	**Decryption** Implements inverse byte substitution only
Column Mix	**Encryption** Main substitution box - combines the byte substitution and column mix operations
Inverse Mix	**Decryption** Inverse substitution box - combines the byte substitution and inverse column mix operations
Key Mix	**Keying** Column mix box for computing the inverse key table

These tables are all derived from the basic GF(2^8) mathematics outlined in the specification. A simple C program is used to generate all tables and print them as C array declarations to compile and link with the algorithm codes. While 20K bytes of tables may be not optimal for some target implementations, large memory, large cache machines like PA-RISC and IA-64 gain substantial performance with what is negligible extra data. Rijndael outperforms all other AES submissions in keying, encryption, and decryption. In particular, Rijndael keying is a full order of magnitude faster than most other algorithms.

Keying

Rijndael key expansion looks largely serial. There are four look-ups every fourth key word, but little else to suggest parallelism. The discussion in section 5.3.3, however, shows that decryption can be more efficiently implemented if an "inverse" key table is used. If the basic key generation loop is unrolled four times, we can combine the inverse key computation with the key generation:

```
A = SubByte(RotByte(D)) ^ Rcon[i];
B = B ^ A;
C = C ^ B;
D = D ^ C;
IA = InvMixColumn(A);
IB = InvMixColumn(B);
IC = InvMixColumn(C);
ID = InvMixColumn(D);
```

Clearly, the `InvMixColumn` operation, which is four byte-indexed lookups into four 256-entry tables and three XORs, can begin as soon as the key word is ready. Thus, both the forward and inverse key tables can be computed in the same time as computing the inverse table. As a minor space optimization, the last forward key and first inverse key, which are identical, are stored only once in a combined key table.

Both the PA-RISC and IA-64 implementations are straightforward: as soon as the forward key is available, start the look-ups for the inverse key. Two look-ups are performed on the key word `A`, but only one set of byte extractions is needed, saving four operations per

round. PA-RISC has 28 registers available to a subroutine: all of these are needed to hold the intermediate results. The large register file on IA-64 provides enough temporary registers to perform the computation with maximum concurrency. Rijndael keying improves greatly when everything can be kept in registers.

On PA-RISC, a load address can be the sum of a base register and a scaled offset register; thus, table look-up requires two instructions. IA-64, however, only takes a load address from a register without offset. Therefore, a table look-up must explicitly scale the index and add it to the desired table address: this is accomplished with the `shladd` instruction. The sequence of extract, scale and add, load is pipelined, so that the entire look-up sequence only requires one extra cycle over the equivalent PA-RISC sequence. The greater parallelism in IA-64 allows the forward key computation and XOR trees to overlap the look-ups, giving it an overall performance advantage.

Encryption

Rijndael encryption, while defined as several, separate steps, can be collapsed into a single set of table look-ups by (1) computing the look-up tables to combine the byte substitution and column mix operations, and (2) selecting the index bytes from the data block to reflect the row rotation in each round. Decryption is identical except for the look-up table and the order of byte selection. It is not surprising, then, that encryption and decryption are very similar to keying, except that only 16 look-ups are done per round instead of the 20 performed for each keying round.

Performance

Cycles	Pentium	PA-RISC			IA-64	IA-64++
		Min	Average	Max		
Keying	1338	239	249.25	261	148	148
Forward Keying	217	85	92.18	101	104	104
Encryption	284	168	175.5	193	124	124
Decryption	283	168	175.88	192	125	125

On PA-RISC, Rijndael full keying executes in 239 cycles, compared to the best-reported Pentium results of 1338, a 5.6:1 performance advantage. Encryption and decryption are faster: 40.9% faster for encryption (168 vs. 284) and 40.6% faster for decryption (168 vs. 283). On IA-64, keying completes in 148 cycles, a 9:1 performance improvement over Pentium. Encryption and decryption are also faster: 56.3% faster for encryption (124 vs. 284) and 55.8% faster for decryption (125 vs. 283).

The parallelism of Rijndael saturates a two-banger IA-64. To explore the limits of Rijndael's parallelism, a code schedule was developed for a hypothetical, four-banger implementation. With this 12-way parallel IA-64, the inner loop of Rijndael encryption can be executed in 7 cycles, which suggests a total encryption time of 74 cycles per 128-bit data block. This is only one cycle short of the theoretical limit of 6 cycles per round for an arbitrarily wide IA-64 implementation, which would perform 16 extracts, 20 address computations, 20 loads, then three levels of XORs.

Serpent

The heart of the Serpent algorithm (Anderson, Biham, Knudsen) is the set of Boolean equations implementing the "bit-slice" substitution boxes. One set of equations was submitted with the AES proposal; Brian Gladman and Sam Simpson used a recursive expression search program to develop an alternative set of equations that improved performance on the Pentium-II platform. Dr. Gladman, however, cautions on his Serpent web page[4]:

> *"On any particular machine it will be desirable to experiment with the order of terms (where there is quite a lot of flexibility) and with the reuse of the temporary variables used during function evaluation."*

Taking this advice to heart, the two sets of equations, along with an earlier version of Gladman's equations, and a set of equations optimized for Pentium submitted to the authors by Dag Arne Osvik[5], were analyzed according to the following metrics:

Ops Count of Boolean operations required to compute the substitution or reverse substitution function. The equation parser looks for occurrences of `A & ~B` to take advantage of the `and-complement` instruction in both the PA-RISC and IA-64 instruction sets.

Cycles Number of steps required to complete the computation on a highly parallel machine, such as IA-64, and a two-ALU operation superscalar machine, such as PA-RISC.

Width For IA-64, the largest number of operations executed concurrently.

Temps Number of temporary values. In order to reduce the number of temporaries, a simple register analysis was performed that first re-used the output terms as intermediate results, then assigned temporaries as needed by the computation.

The results of this analysis for IA-64, summarized in Table 1 below, are interesting: even though the Gladman equations consistently have fewer operations than the others, only 4 of the 16 sets compute faster. When the equations are analyzed for two-ALU

[4] The expression search program, Boolean equations and reference implementations are available at **http://www.btinternet/~brian.gladman/cryptography_technology/Serpent**

[5] Dag Arne Osvik **osvik@ii.uib.no**

operation on PA-RISC, the results (Table 2) favor Gladman's equations, but four of Osvik's equations compute faster. A follow-up submission from Mr. Osvik for S-Box 3 resulted in a spectacular, 4-cycle, solution for IA-64, even though it has the highest operation count of any equation.

The conclusion here is that there is no optimal set of bit-slice equations for all Serpent implementations: the capability and constraints of the target machine must be carefully considered. The authors invite others to submit their own equations for analysis, and offer the analysis tools used here to the Serpent team for their own use.

Keying

Serpent keying starts with the input key, padded to 256 bits, and generates 132 4-byte values with the recurrence:

$$W_i = (W_{i-8} \oplus W_{i-5} \oplus W_{i-3} \oplus W_{i-1} \oplus \Phi \oplus i) <<< 11$$

where W_{-8} = input key word 0, W_{-7} = input key word 1, etc., and Φ is `0x9e3779b9`, derived from the Golden ratio. The resulting values, $[W_0 \ldots W_{131}]$, are then processed in groups of four, $<W_n, W_{n+1}, W_{n+2}, W_{n+3}>$, applying the Serpent forward substitution boxes in the order $S_3, S_2, S_1, S_0, S_7, \ldots, S_4, S_3$. This generates the 33 128-bit keys required for encryption.

Inspecting the recurrence, there is an active state of eight words and that W_i replaces W_{i-8} at each step. If we label the initial key words W_{-8} = A, W_{-7} = B, … W_{-1} = H, we can rewrite the recurrence as the following pattern:

```
A′ = (A ⊕ D  ⊕ F  ⊕ H  ⊕ Φ ⊕ 0) <<< 11
B′ = (B ⊕ E  ⊕ G  ⊕ A′ ⊕ Φ ⊕ 1) <<< 11
C′ = (C ⊕ F  ⊕ H  ⊕ B′ ⊕ Φ ⊕ 2) <<< 11
D′ = (D ⊕ G  ⊕ A′ ⊕ C′ ⊕ Φ ⊕ 3) <<< 11
E′ = (E ⊕ H  ⊕ B′ ⊕ D′ ⊕ Φ ⊕ 5) <<< 11
F′ = (F ⊕ A′ ⊕ C′ ⊕ E′ ⊕ Φ ⊕ 6) <<< 11
G′ = (G ⊕ B′ ⊕ D′ ⊕ F′ ⊕ Φ ⊕ 7) <<< 11
H′ = (H ⊕ C′ ⊕ E′ ⊕ G′ ⊕ Φ ⊕ 8) <<< 11
                  . . .
A′ = (A ⊕ D  ⊕ F  ⊕ H  ⊕ Φ ⊕ 128) <<< 11
B′ = (B ⊕ E  ⊕ G  ⊕ A′ ⊕ Φ ⊕ 129) <<< 11
C′ = (C ⊕ F  ⊕ H  ⊕ B′ ⊕ Φ ⊕ 130) <<< 11
D′ = (D ⊕ G  ⊕ A′ ⊕ C′ ⊕ Φ ⊕ 131) <<< 11
```

This formulation has some limited parallelism in the `XOR` trees. Eventually, the equations will serialize on the 11-bit rotation, but the overall sequence can be organized on a parallel machine to minimize the performance effect. Intermediate loads and stores can be eliminated by overlapping the S-box lookup for $<W_n, W_{n+1}, W_{n+2}, W_{n+3}>$ with the computation of $<W_{n+4}, W_{n+5}, W_{n+6}, W_{n+7}>$. Because different S-boxes are used at each step, the highest performance for Serpent keying is realized by a straight-line implementation.

On PA-RISC, limited to two-way integer instruction parallelism, each set of four recurrence computations saturates the processor for 11 cycles (22 operations). The 11-bit rotation is implemented with a single instruction (`shrpw`); common subexpressions (e.g., `F ⊕ H`) remove two of the 24 operations (five XORs and one rotate per step, times four steps). Since PA-RISC does not have an immediate XOR operation, the `(Φ ⊕ i)` term is computed by adding the low 11 bits of the value (constant for each step) to the high 21 bits (constant for all steps); thus, the computation still occurs in one cycle. To avoid errors, the 11-bit values are generated by a simple program.

IA-64 rotation requires two instructions (deposit and shift register pair). This increases the cycle count for computing four steps from 11 on PA-RISC to 14. However, the machine's greater parallelism can be employed to overlap S-Box and recurrence logic as follows:

```
Recurrence(W0, W1, W2, W3)
Recurrence(W4, W5, W6, W7)          Sbox3(W0, W1, W2, W3)
Recurrence(W8, W9, W10, W11)        Sbox2(W4, W5, W6, W7)
Recurrence(W12, W13, W14, W15)      Sbox1(W8, W9, W10, W11)
                  . . .                       . . .
Recurrence(W124, W125, W126, W127)  Sbox5(W120, W121, W122, W123)
Recurrence(W128, W129, W130, W131)  Sbox4(W124, W125, W126, W127)
                                    Sbox3(W128, W129, W130, W131)
```

Each step in this parallel evaluation, including storing the key words, executes in the 14 cycles needed for the recurrence alone, yielding a substantial speed-up for Serpent keying.

Encryption

Serpent encryption and decryption use 32 rounds of key exclusive OR's, substitution box logic and linear transforms. The S-box issues are almost identical to those for keying, as discussed above. The linear transform, which accelerates the avalanche effect, limits the potential for overlap with the S-box computations. Depending on the S-box equations used, at most one or two cycles can be removed per S-box; the current implementation overlaps one cycle for six of the eight S-box equations.

The forward linear transform, diagrammed in Figure 1, consists of 16 operations (six fixed rotations, two rotations, eight exclusive-OR's). Ideally, this sequence can be executed in seven cycles on a parallel machine:

```
X0 = X0 <<< 13      X2 = X2 <<< 3
X1 = X1 ⊕ X0        X3 = X3 ⊕ X2        T1 = X0 << 3
X1 = X1 ⊕ X2        X3 = X3 ⊕ T1
X1 = X1 <<< 1       X3 = X3 <<< 7
X0 = X0 ⊕ X3        X2 = X2 ⊕ X3        T2 = X1 << 7
X0 = X0 ⊕ X1        X2 = X2 ⊕ T2
X0 = X0 <<< 5       X2 = X2 <<< 22
```

The inverse linear transform, diagrammed in Figure 2, also has 16 operations; however, it can be computed in five cycles:

```
X0 = X0 >>> 5       X2 = X2 >>> 22      T1 = X1 ⊕ X3        T2 = X3 >>> 7
X0 = X0 ⊕ T1        X2 = X2 ⊕ X3        T3 = X1 << 7        X1 = X1 >>> 1
X1 = X1 ⊕ X0        X2 = X2 ⊕ T3        T4 = X0 << 3
X1 = X1 ⊕ X2        X3 = X3 ⊕ X2        X0 = X0 >>> 13
X3 = X3 ⊕ T4        X2 = X2 >>> 3
```

On PA-RISC, the single-cycle fixed rotation allows both transforms to execute in eight cycles, optimal for the two-way superscalar machine. The two-cycle rotation on IA-64 increases the operation count to 22, and the dependencies are such that the best implementation for the transforms requires 12 cycles. Loading and XORing the key material in parallel with the transforms can reclaim some performance; however, the linear transformation accounts for over 50% of the encryption and decryption cycles.

As with keying, the best performance is achieved with straight-line code. The program source for both PA-RISC and IA-64 make heavy use of macros and bear strong resemblance to the algorithm specification. An extension of the software tools used to analyze Serpent equations actually produces the raw instruction stream for each equation, in either machine language format, which is then easily integrated into the source program through the macro definitions.

Performance

Cycles	Pentium	PA-RISC			IA-64	IA-64++
		Min	Average	Max		
Keying	1292	668	668.79	669	475	380
Encryption	900	580	580	580	565	468
Decryption	885	585	586.62	587	631	407

On PA-RISC, Serpent keying executes in 668 cycles, compared to the best-reported Pentium results of 1292, almost a 2:1 performance advantage. Encryption and decryption also run substantially faster: a 35.6% advantage for encryption (580 vs. 900) and a 33.9% advantage for decryption (585 vs. 885).

On IA-64, the extra parallelism pays off handsomely in keying, where the routine completes in 475 cycles, a 2.7:1 performance gain over Pentium. Encryption and decryption, with the extra cycles required to complete the linear transform, are better than Pentium, although not as overwhelmingly: 37.2% for encryption (565 vs. 900), 28.7% for decryption (631 vs. 885). For IA-64++, keying is estimated to run in 380 cycles, 3.4 times faster than Pentium, encryption in 468 cycles (48.0% faster), and decryption in 407 cycles (54% faster).

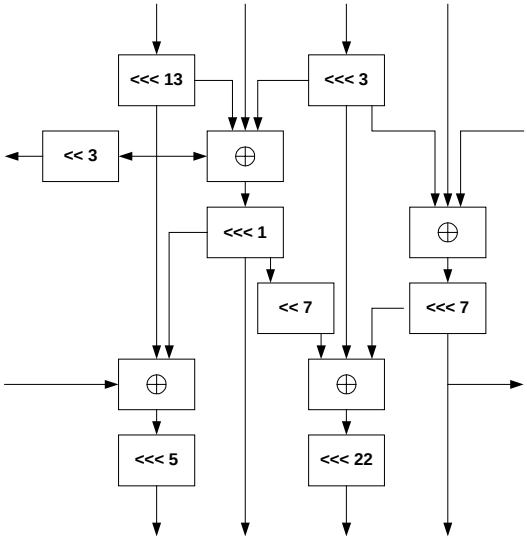


Figure 1 – Serpent Linear Transform

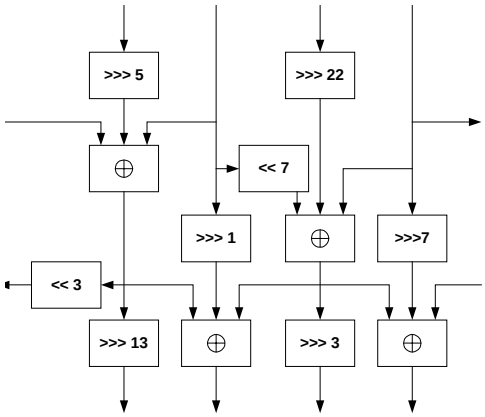


Figure 2 – Serpent Inverse Transform

	AES Submission				Gladman (Best)				Osvik			
	Ops	**Cycles**	**Width**	**Tmps**	**Ops**	**Cycles**	**Width**	**Tmps**	**Ops**	**Cycles**	**Width**	**Tmps**
S Box 0	18	9	6	4	15	6	4	3	17	6	4	5
S Box 1	18	9	5	3	14	8	3	2	17	7	3	3
S Box 2	16	9	3	4	16	8	3	3	14	7	3	5
S Box 3	18	7	5	4	16	8	5	3	21	4	6	6
S Box 4	19	7	4	5	15	8	3	3	19	9	3	3
S Box 5	17	8	3	4	16	7	4	3	18	7	3	3
S Box 6	19	6	7	4	17	6	5	4	17	9	3	3
S Box 7	19	8	4	3	17	11	3	3	19	8	4	5
I Box 0	19	8	5	4	15	10	2	2	18	8	3	4
I Box 1	18	9	3	3	17	7	5	2	18	11	3	3
I Box 2	18	7	5	4	16	8	4	3	18	7	3	3
I Box 3	17	7	4	3	17	9	4	4	17	8	3	3
I Box 4	17	7	4	4	17	6	5	5	19	11	3	3
I Box 5	17	7	5	4	16	7	4	3	18	10	2	3
I Box 6	19	6	4	4	17	8	4	2	16	8	3	3
I Box 7	18	9	4	2	17	9	3	2	18	8	4	4

Table 1 - Serpent IA-64 Metrics

	AES Submission			Gladman (Best)			Osvik		
	Ops	**Cycles**	**Tmps**	**Ops**	**Cycles**	**Tmps**	**Ops**	**Cycles**	**Tmps**
S Box 0	18	11	3	15	8	2	17	9	2
S Box 1	18	11	3	14	8	2	17	9	3
S Box 2	16	11	4	16	9	3	14	8	3
S Box 3	18	9	4	16	9	3	17	9	3
S Box 4	19	10	6	15	8	3	19	10	1
S Box 5	17	9	4	16	9	3	18	9	1
S Box 6	19	10	4	15	9	3	17	10	2
S Box 7	19	10	3	17	12	5	19	10	2
I Box 0	19	10	4	15	10	2	18	11	2
I Box 1	18	10	3	17	9	3	18	11	2
I Box 2	18	10	3	16	9	2	18	10	2
I Box 3	17	9	3	17	9	4	17	9	1
I Box 4	17	9	5	17	9	4	19	11	2
I Box 5	17	9	4	16	8	4	18	10	2
I Box 6	19	10	5	17	9	3	16	8	2
I Box 7	18	9	3	17	10	2	18	9	2

Table 2 - Serpent PA-RISC Metrics

Twofish

The Twofish block cipher employs a *"Feistel-like structure with additional whitening of the input and output."*[6] The 128-bit plaintext block is split into four 32-bit words. In the input whitening step each 32-bit word is XORed with a different 32-bit input-whitening key. This is followed by 16 rounds in which the left two words are transformed by the F-function. The leftmost word produced by the F-function is XORed with the third word, and the result is rotated to the right by one bit. The rightmost word produced by the F-function is XORed with the fourth word, which previously had been rotated to the left by one bit. For all but the 16th round, the left and right pairs of words then are swapped for the next round. Each of the final four words is XORed with a different 32-bit output-whitening key.

Within the F-function, the first input word is transformed by the g-function. The second input word first is rotated to the left by eight bits, and then transformed by the g-function. The two g-function outputs then are mixed into two new words by a Pseudo-Hadamard Transform (PHT). After mixing, a different round key is added to each of the two new words, producing the two output words of the F-function.

The g-function may be implemented in a variety of ways, depending upon one's choice of keying strategy. Twofish defines five different keying strategies: Compiled, Full, Partial, Minimum, and Zero. These choices enable a wide range of time/memory trade-offs for a Twofish implementation.

For RISC and EPIC microprocessors, the choice of Full keying is the most natural. Full keying requires 4096+128+32 = 4256 bytes of table for the four key-dependent S-boxes, 32 round keys, and eight whitening keys. This table size poses no problem for a modern computer platform. Compiled keying is able to reduce the Twofish Pentium-Pro encryption time from 315 cycles to 258 cycles, but it necessitates a separate copy of the encryption and decryption codes for each different key. For superscalar RISC and EPIC microprocessors, Compiled keying is unlikely to result in a performance gain. Given sufficiently many general registers, key loading always can be overlapped and executed in parallel.

The heart of the Twofish g-function is defined as:

1. Partition the 32-bit input word into four 8-bit bytes.
2. Use the value of each of the four bytes to index and fetch a new byte value from a corresponding, 256 byte, key-dependent S-box.
3. Matrix multiply the MDS matrix, a predefined, maximal distance separation byte matrix by the vector of the four bytes fetched from the S-boxes. Scalar multiply of bytes in $GF(2^8)$ is represented as GF(2)[x] modulo v(x), where v(x) is the primitive polynomial $x^8+x^6+x^5+x^3+1$. Scalar addition of bytes in $GF(2^8)$ is XOR.

For Full keying, each of the four S-boxes contains 256 32-bit words, rather than 256 8-bit bytes. Each 32-bit word of $S\text{-box}_{32}[i]$ is the four-byte vector computed by matrix multiplication of the MDS matrix by the four-byte vector whose sole non-zero component is the byte $S\text{-box}_8[i]$. If we denote matrix multiplication by +.×, and the bytes of a column vector, least significant byte first, as [B0:B3] or [B0, B1, B2, B3] the 32-bit S-boxes are:

$$S\text{-box}0_{32}[i] = MDS +.\times [S\text{-box}0_8[i], 0, 0, 0]$$
$$S\text{-box}1_{32}[i] = MDS +.\times [0, S\text{-box}1_8[i], 0, 0]$$
$$S\text{-box}2_{32}[i] = MDS +.\times [0, 0, S\text{-box}2_8[i], 0]$$
$$S\text{-box}3_{32}[i] = MDS +.\times [0, 0, 0, S\text{-box}3_8[i]]$$

In this manner, all $GF(2^8)$ byte multiplications of the g-function MDS matrix multiply are pre-computed, and saved in the 32-bit S-boxes. With these S-boxes, all that is required for a g-function MDS matrix multiplication is to fetch a 32-bit word from each of the four S-boxes and XOR the words together. Therefore, the Full keying computation of the g-function consists of extracting four 8-bit bytes from the input word, using each extracted byte to index and fetch a 32-bit word from a corresponding S-box, and XORing the four fetched words. The rotation by eight bits of the right input word to the F-function actually requires no explicit computation. It is accomplished simply by the order in which 8-bit bytes are extracted from the input word. Similarly, no computation is required for word swapping between rounds.

Keying

Full keying for a Twofish 128-bit user-supplied key proceeds in three phases. In each phase the approach taken utilizes modestly sized tables to accelerate the performance. The user-supplied key is taken as four 32-bit words, in little-endian byte order. These words are called M_0, M_1, M_2, and M_3. Their byte contents, respectively are: $[m_0:m_3]$, $[m_4:m_7]$, $[m_8:m_{11}]$, and $[m_{12}:m_{15}]$, where m_i is the i'th byte of the user-supplied key.

In the first phase of keying, two four-byte vectors denoted S_0 and S_1 are derived from the user-supplied key. These vectors are utilized in the computation of the S-boxes. S_0 and S_1 each are computed by a matrix multiplication of the RS matrix by an eight-byte vector of user-supplied key bytes. The 4×8 RS matrix is derived from a Reed-Solomon code, and is specified by the Twofish definition. Specifically:

$$S_0 = [RS] +.\times [m_0:m_7] \qquad S_1 = [RS] +.\times [m_8:m_{15}]$$

For the RS matrix multiplication, scalar multiply of bytes in $GF(2^8)$ is represented as GF(2)[x] modulo w(x), where w(x) is the primitive polynomial $x^8+x^6+x^3+x^2+1$. Scalar addition of bytes in $GF(2^8)$ is XOR. The actual computation of these two matrix multiplications is accomplished by simulating the LFSRs for the RS code. Doug Whiting programmed this in the following manner in the original Twofish submission.

---

[6] Schneier, Kelsey, Whiting, Wagner, Hall, Ferguson, *The Twofish Encryption Algorithm*, John Wiley & Sons, 1999.

```
#define      RS_GF_FDBK   0x14D        /* field generator */
#define      RS_rem(x)                                                  \
{ BYTE  b  = x >> 24;                                                   \
  DWORD g2 = ((b << 1) ^ ((b & 0x80) ? RS_GF_FDBK : 0 )) & 0xFF;        \
  DWORD g3 = ((b >> 1) & 0x7F) ^ ((b & 1) ? RS_GF_FDBK>>1 : 0) ^ g2;    \
        x = (x << 8) ^ (g3 << 24) ^ (g2 << 16) ^ (g3 << 8) ^ b;
}
```

S_0 and S_1 then can be calculated by the following triply-nested loop, where `M[i]` denotes M_i and `S[i]` denotes S_i:

```
for( i = 0; i < 2; ++i ) {
    for( j = 0, r=0; j < 2; ++j ) {
        r ^= (j) ? M[i*2] : M[i*2+1];
        for( k = 0; k < 4; ++k ) {
            RS_rem( r );
        }
        S[i] = r;
    }
}
```

The calculation of S_0 and S_1 can be accelerated by using a pre-computing a table of 32-bit words, `RStbl[256]`, where `RStbl[i]` = `RS_prem(i)`. `RS_prem(x)` is identical to `RS_rem(x)` but without the `(x << 8)` term in the final assignment statement. Each cycle of the LFSRs then may be simulated simply by:

```
unsigned int x;
#define   RS_rem(x)  x = (x << 8) ^ RStbl[x >> 24];
```

The triply-nested loop to compute S_0 and S_1 is completely unrolled. Housekeeping instructions may be executed in parallel with this computation.

The second phase of keying is to compute the four key-dependent S-boxes. Four pre-computed, 256 entry, 32-bit word auxiliary tables are utilized to accelerate this computation. These tables, denoted `MD0`, `MD1`, `MD2`, and `MD3`, are similar to the Full key S-boxes. Two additional 256 entry, 8-bit tables are required for the S-box computation. These are the tables containing the basic q0 and q1 byte permutations defined in the Twofish specification. These tables are denoted `q0` and `q1`. Each auxiliary table entry combines the final q0 or q1 byte permutation of the S-box computation, and the MDS matrix multiplication. Specifically :

```
MD0[i]  =  MDS +.× [q1[i], 0, 0, 0]
MD1[i]  =  MDS +.× [0, q0[i], 0, 0]
MD2[i]  =  MDS +.× [0, 0, q1[i], 0]
MD3[i]  =  MDS +.× [0, 0, 0, q0[i]]
```

This is the same matrix multiplication used in the g-function. Each Full key S-box contains exactly the same 32-bit words as the corresponding auxiliary table, but permuted according to the user-supplied key. If we designate the bytes of the words S_0 and S_1 as S0(3:0) and S1(3:0), byte zero being least significant, the Full key S-box computation loop is:

```
for( i = 0; i < 256; ++i ) {
   S-box0₃₂[i] = MD0[ q0[ q0[i]^S0(0) ] ^ S1(0) ];
   S-box1₃₂[1] = MD1[ q0[ q1[i]^S0(1) ] ^ S1(1) ];
   S-box2₃₂[i] = MD2[ q1[ q0[i]^S0(2) ] ^ S1(2) ];
   S-box3₃₂[i] = MD3[ q1[ q1[i]^S0(3) ] ^ S1(3) ];
}
```

This computation further can be accelerated by yet another, 256-entry, auxiliary 32-bit word table. This table is called `q0q1q0q1`. The i'th entry of this table consists of `[ q0[i], q1[i], q0[i], q1[i] ]`. The word `q0q1q0q1[i]` can be fetched by a single instruction, and can be XORed with S_0. This computes the inner XOR of all four assignment statements in parallel. Each byte of this intermediate result then is used to fetch a byte from q0 or q1. Following one more XOR with the corresponding byte of S_1, the $S\text{-box}_{32}$ entry is obtained by indexing and fetching the 32-bit word from the proper MD table. This word is stored into the proper 32-bit S-box.

The code to perform this computation is organized as a 256-pass loop for both PA-RISC and IA-64. The S_0 and S_1 words already reside in general registers. For each loop iteration, the required operations are one indexed load for the q0q1q0q1 table entry, a word XOR with S_0, four byte extracts[7], four indexed byte loads from the q0 and q1 tables, four XORs with S_1 bytes, four indexed word loads from the MD tables, four indexed word stores to the S-boxes, and a loop closing instruction. For IA-64, eight additional instructions are required for computing table addresses. IA-64 post address modification is used for indexing the q0q1q0q1 table and the S-boxes.

The total number of 256-entry tables used to accelerate the computation of S_0, S_1, and the key-dependent S-boxes is eight, occupying 6656 bytes. These table sizes are quite acceptable for a modern RISC or EPIC platform. No IA-64 bank optimization was done for these tables[8]. No additional tables are required for the third phase of keying.

1. q0 256 bytes
2. q1 256 bytes
3. q0q1q0q1 1024 bytes
4. MD0, …, MD3 1024 bytes each, 4096 bytes total
5. RStbl 1024 bytes

[7] The four S_1 byte extracts are done outside the loop.

[8] Described in the next section.

The third and final phase of keying is the computation of the 40 whitening and round keys. This code is similar to the computation of the S-boxes. It is organized as a 20-iteration loop, in which two keys are computed per iteration. Unlike the S-box computation, each key requires a full MDS matrix multiply. Further, a final PHT transform is applied to each pair of keys. The Twofish definition systematically uses the same MDS matrix multiply and PHT operations in the keying algorithms and in the encryption and decryption algorithms.

The same table techniques used above are used to accelerate computation of the whitening and round keys. The initial eight of the 40 keys are taken as the input and output whitening keys. The final 32 keys are taken as the round keys. Using the previously defined notations, and K to denote the newly computed keys, the computation for the 40 whitening and round keys is:

```
for( i = 0; i < 40; i += 2 ) {
   T0  = MD0[ q0[ q0[i]^M2(0) ] ^ M0(0) ];
   T0 ^= MD1[ q0[ q1[i]^M2(1) ] ^ M0(1) ];
   T0 ^= MD2[ q1[ q0[i]^M2(2) ] ^ M0(2) ];
   T0 ^= MD3[ q1[ q1[i]^M2(3) ] ^ M0(3) ];
   T1  = MD0[ q0[ q0[i+1]^M3(0) ] ^ M1(0) ];
   T1 ^= MD1[ q0[ q1[i+1]^M3(1) ] ^ M1(1) ];
   T1 ^= MD2[ q1[ q0[i+1]^M3(2) ] ^ M1(2) ];
   T1 ^= MD3[ q1[ q1[i+1]^M3(3) ] ^ M1(3) ];
   T1  = (T1 <<< 8);
   T0 += T1;
   T1 += T0;
   T1  = (T1 <<< 9);
   K[i] = T0;
   K[i+1] = T1;
}
```

The code to perform this computation is organized as a 20-pass loop for both PA-RISC and IA-64. Note that the M_i words are used in even-subscript and odd-subscript pairs. Also note that the M_i words are used in an order reversed from the order of the S_i words in the S-box computation. The M_0, M_1, M_2, and M_3 words already reside in general registers. For each loop iteration, the required operations are two indexed loads for the q0q1q0q1 table entries, two word XORs with M_2 and M_3, eight byte extracts[9], eight indexed byte loads from the q0 and q1 tables, eight XORs with M_0 and M_2 bytes, eight indexed word loads from the MD tables, six XORs to complete the MDS matrix multiplies, two rotates, one add and one shift-and-add for the PHT, two indexed word stores to the key array, and a loop closing instruction. For IA-64, sixteen additional instructions are required for computing table addresses. IA-64 post address modification is used for indexing the q0q1q0q1 table and the key array.

Encryption

For PA-RISC the encryption and decryption functions are organized as straight-line code. Each is provided two pointer arguments, the first to the 16-byte cleartext block or ciphertext block, the second to the concatenation of the round keys, whitening keys, and four Full key S-boxes. Input blocks are whitened 64-bits at a time. Housekeeping instructions are overlapped with the first and last rounds.

Each PA-RISC round, including the one-bit circular shifts, executes in about a dozen cycles. PA-RISC includes an instruction that can extract any contiguous 8-bit field from a word in one cycle. The extracted byte can be used directly as an index for a 32-bit word load instruction. Further, the PA-RISC shift-and-add instruction permits the PHT to be done in two instructions during the same cycle. Thus, each round needs 32 instructions: eight extract instructions (`extrw,u`), eight instructions to load from S-boxes (`ldw,s`), two instructions to load round keys (`ldw`), two one-bit circular shift instructions (`shrpw`), eight XOR instructions (`xor`), three add instructions (`add,l`), and one shift-and-add instruction (`shladd,l`). The instruction schedule is nearly optimal, but the final right rotate by one bit adds one cycle to the round.

For IA-64 the encryption and decryption functions are organized in exactly the same way. In each round, an additional instruction is required to compute an S-box address from each extracted byte. Although this requires eight additional instructions, there also is an added benefit. Microprocessor caches often are organized as independent 8-byte banks. An optimal memory strategy, therefore, shuffles the four S-boxes, so that each S-box is entirely contained in a single cache bank. This results in a 16-byte stride between successive S-box words. The IA-64 shift-and-add instructions, used to compute S-box addresses, therefore, use a shift value of four. This assures the absence of cache bank conflicts when executing two S-box loads during the same cycle.

A second technique employed for IA-64 is computational height reduction, a practice common for parallel instruction issue machines. Additional instructions are executed, but the entire computation completes in fewer cycles.

In Twofish, the rightmost bit of the first F-function output becomes the high order bit of a byte to be extracted in the next round. For PA-RISC, the fact that the extract instruction demands a contiguous bit field requires that the one-bit right rotate be done after computation of the first F-function output and prior to the extract for the next round. For IA-64, parallelism and predicates offer a better solution.

The first F-function output is computed as three XORs, two adds, and a final XOR. Although these operations do not commute or freely associate, they in fact do so for the rightmost bit, which actually is the result of six XORs. By computing the rightmost bit of the last XOR sooner (round-key XOR third-block-word), one redundantly can compute the rightmost bit of the first F-function output one cycle earlier. This permits the rightmost bit also to be tested without adding a cycle to the round. The result of the test is written to a predicate. This predicate then is used to set a temporary S-box pointer either to the beginning, or to the halfway point, of the corresponding S-box at the start of the next round. Only the seven leftmost bits of the unrotated first F-function output are extracted in

[9] The eight M_0 and M_1 byte extracts are done outside the loop.

the next round. They then are used as an index relative to the temporary pointer. The full first F-function output word can be rotated later.

It also turns out that, with proper table alignment, height reduction can be used to compute two S-box addresses one cycle earlier in the next round. The enabling fact here is that offsets into S-boxes consist of 12 bits, of which the right four are zero. For a 4096-byte aligned and shuffled table, an XOR can be used for the address calculation. The terms for two such XORs redundantly can be computed in the previous round. This can be seen from the following equations for one pair of encryption terms (note: [7:0] denotes the rightmost 8 bits of a word):

```
Let:   PHT1 be the second PHT output for the current Round.
RK1    be the second Key word for the current Round.
BW3    be the fourth Block word for the current Round.
Fin1   be the second input word to the next Round.
PSB1   be the pointer to S-box 1.
pSBE   be the pointer to the S-box 1 entry for Fin1[7:0].[10]

Fin1   = (PHT1+RK1) ⊕ BW3

pSBE   = pSB1  +  16*( Fin1 )[7:0]
       = pSB1  +  16*( (PHT1+RK1) ⊕ BW3 )[7:0]
       = pSB1  +  ( 16*(PHT1+RK1)[7:0] ⊕ 16*BW3[7:0] )
       = pSB1  ⊕  ( 16*(PHT1+RK1)[7:0] ⊕ 16*BW3[7:0] ) [11]
       = (pSB1 ⊕ 16*BW3[7:0]) ⊕ (16*(PHT1+RK1)[7:0])
```

Performance

Cycles	Pentium	PA-RISC			IA-64	IA-64++
		Min	Average	Max		
Keying	8414	2846	2901.79	2964	2445	2445
Encryption	315	205	217.45	233	182	182
Decryption	311	200	210.29	224	182	182

On PA-RISC, Twofish keying executes in 2846 cycles, compared to the best-reported Pentium results of 8414, a 2.96:1 performance advantage. Encryption and decryption also run faster: a 36% advantage for encryption (205 vs. 315) and a 35.7% advantage for decryption (200 vs. 311).

On IA-64, Twofish executes even faster. Twofish keying executes in 2445 cycles, compared to the best-reported Pentium results of 8414, a 3.44:1 performance advantage. Encryption and decryption also run faster: a 42.2% advantage for encryption (182 vs. 315) and a 41.5% advantage for decryption (182 vs. 311).

[10] S-box 1 is used for the rightmost bits because of the logical (Fin_1 <<< 8).

[11] Addition is equivalent to exclusive-or because of the S-box table alignment.

Conclusions

All the algorithms have reasonable implementations on PA-RISC and IA-64; all make good use of the architectures. It is clear that the underlying computer architecture has a direct and significant effect on the optimal implementation for each candidate. The large register files in PA-RISC and IA-64 enable complete state to be kept without using memory, influencing the structure of Rijndael, Twofish, and keying codes. The choice of equations for Serpent is a direct result of the available execution width and ALU operations. Sometimes, effects are expressible only at the assembly level, such as the software pipelines in the Mars keying or the MMU multiplication in RC6 encryption. In other cases, algorithm structures to exploit the underlying architecture are best expressed in high level source, such as the restructuring of the RC6 keying algorithm.

Our second conclusion is that *algorithm performance cannot be measured by a single number*. A complete performance characterization must filter out large system effects such as caching, memory latencies, interrupts, paging, process swaps, and I/O activity, but should draw attention to fine-grain system effects such as cache interference and execution latencies. When timing keying for random input key values, the results will exhibit a performance distribution rather than a single number.

Another consideration is parallelism. Future CPU's will be increasingly, and we believe explicitly, parallel; algorithms that can exploit parallelism will see continuing performance improvement over the life of the new AES algorithm. It should be observed that as better Serpent equations are developed, Serpent will further improve both its performance and parallelism. A final factor in evaluating software is memory usage; none of the finalists use tables uncomfortably large for modern server and desktop systems.

Using these criteria, and assuming that the IA-64++ additions will/will-not be made, the results of this study rank the AES finalists as follows:

Performance	Memory	Parallelism
Rijndael	RC6	Rijndael
RC6/Twofish	Serpent	Twofish
Twofish/RC6	Mars	Serpent
Mars	Twofish	Mars
Serpent	Rijndael	RC6

Acknowledgments

We wish to express our thanks to Doug Whiting for his unerring guidance, especially his prescient counsel in the selection of first-round candidates to investigate. We also wish to thank and to acknowledge the contributions of Dr. Brian Gladman, whose work is well known and appreciated by the AES community. Brian's codes were used to generate test values and, in many instances, improved our understanding of the algorithms. Brian also kept us up to date on his Pentium performance improvements. We appreciate Rohit Bhatia's suggestions for 32×32 multiplies. Dag Arne Osvik contributed his Serpent equations, which forced our equation analysis tools to improve and sped up both the PA-RISC and the IA-64 Serpent implementations.

We are indebted to John Crawford and members of the Intel Itanium team, who provided access and support for the Itanium simulator. Finally, our thanks go to the Hewlett Packard Ft. Collins McKinley team, whose assistance with the development and simulation tools, and patience with our endless questions, was the *sine qua non* of the IA-64 work.

Appendix A:
Summary of Best Performance

Candidate	Encryption					Decryption					Keying				
	Clocks	Ops	IPC	Regs	Bytes	Clocks	Ops	IPC	Regs	Bytes	Clocks	Ops	IPC	Regs	Bytes
Mars															
Pentium	320					374					3894				
New Keying											2128				
PA-RISC	540	631	1.17	12(18)	2588	538	632	1.17	12(18)	2592	1969	2908	1.48	20	2584
New Keying	538	631	1.17	12(18)	2588	537	632	1.17	12(18)	2592	1797	1805	1.00	20	1984
IA-64	511	1013	1.98	18//8	784	527	1013	1.92	18//8	784	1903	3332	1.75	14//48	1344
New Keying											1408	3132	2.22	12//16	976
IA-64++	255					271					1313				
New Keying											1408				
Table Sizes					2048					2208					2208
Alg Parallelism			2.0					2.0					3.0		
RC6															
Pentium	243					226					1632				
PA-RISC	580	577	0.99	12(4)	2308	493	558	1.13	12(4)	2232	1077	1519	1.41	12	760
IA-64	490	826	1.69	4/27/8	480	490	826	1.69	4/27/8	528	1581	2629	1.66	8//56	256
IA-64++	150					130					1057				
Table Sizes					0					176					176
Alg Parallelism			2.0					2.0					2.0		
Rijndael															
Pentium	284					283					1338				
PA-RISC	168	537	3.20	24	2160	168	539	3.21	24	2160	239	686	2.87	28	2800
Fwd Keying											85	228	2.68	19	1504
IA-64	125	704	5.63	20/12	3808	126	706	5.60	20/12	3824	148	822	5.55	24/21	4480
Fwd Keying											104	282	2.71	19	1504
IA-64++	same					same					same				
Table Sizes					8192					8368					8368
Alg Parallelism			10.0					10.0					10.0		
Serpent															
Pentium	900					885					1301				
PA-RISC	580	1273	2.19	17	5100	585	1309	2.24	17	5240	668	1409	2.11	19	5640
IA-64	565	1517	2.61	24	8480	631	1546	2.45	24	8848	475	1527	3.21	22/4	8368
IA-64++	468					407					380				
Table Sizes					0					528					528
Alg Parallelism			3.0					3.0					4.0		
Twofish															
Pentium	315					311					8414				
PA-RISC	205	548	2.67	20	2192	200	548	2.74	20	2192	2846	8904	3.13	30	1324
IA-64	182	927	5.09	23	5184	182	915	5.03	23	4960	2445	9561	3.91	26/21	1600
IA-64++	same					same					same				
Table Sizes					6656					4256					4256
Alg Parallelism			6.0					6.0					4.0		

Notes:

-- IA64++ is a *hypothetical* IA-64 implementation – refer to the text for details. It does not represent any current or planned IA-64 implementation.

-- Twofish times for Full keying are from: *The Twofish Encryption Algorithm*, John Wiley & Sons, 1999.

-- Pentium, Alpha clocks are lowest reported clocks from the NIST Round 1 Report, August 1999.

-- Regs = GRs, or statics/stacked, or statics//rotating, or statics/stacked/rotating, or GRs(FRs) registers.

-- Bytes are object code sizes. Table Sizes are total tables for *keying*, key table plus look-up tables for *encryption* and *decryption.*

-- Alg Parallelism is an estimated integral upper bound for software parallelism.

Appendix B: Mars Keying
Original Implementation

The original Mars keying initializes the first seven elements of an array, `T[-7..39]`, to the first seven entries of the Mars S-box, then sets the rest of the array as follows:

```
T[i] = ((T[i-7] ⊕ T[i-2]) <<< 3) ⊕ k[i mod N] ⊕ i    i = 1 ... 38
T[39] = N
```

where *k* is the input key and `N` is the size, in words, of the input key. This recurrence has an active state of seven words, `A`, `B`, …, `G`, such that the expansion can be rewritten:

```
T[0]  = A = ((A ⊕ F) <<< 3) ⊕ k[0] ⊕ 0
T[1]  = B = ((B ⊕ G) <<< 3) ⊕ k[1] ⊕ 1
T[2]  = C = ((C ⊕ A) <<< 3) ⊕ k[2] ⊕ 2
T[3]  = D = ((D ⊕ B) <<< 3) ⊕ k[3] ⊕ 3
T[4]  = E = ((E ⊕ C) <<< 3) ⊕ k[0] ⊕ 4
                  . . .
T[38] = D = ((D ⊕ B) <<< 3) ⊕ k[2] ⊕ 38
T[39] = N
```

where `A` is initialized to `S[0]`, `B` to `S[1]`, and so forth. When key expansion is complete, the data words are then "stirred" seven times as follows:

```
T[i] = (T[i] + S[T[i-1] & 0x1ff]]) <<< 9       i = 1 ... 39
T[0] = (T[0] + S[T[39] & 0x1ff]]) <<< 9
```

It is possible to overlap the first stirring with the key expansion: after the first eight expansion steps, `T[1]` is no longer involved in the expansion recurrence and can therefore be stirred. This requires adding one extra word to the expansion state so that both `T[i]` and `T[i-1]` are available for stirring. After the stirring, the keys are reordered, mapping `T[i]` → `K[7i mod 40]`

PA-RISC

The PA-RISC implementation uses straight-line coding for the expansion/stir phase, rotating the key words each step. The remaining six stirring passes are executed in a loop as per the specification. The reordering, however, is again straight-line code. If reordering is considered as replacement rather than a permuted copy, the replacements form chains, that is:

`T[1]` → `T[7]` → `T[9]` → `T[23]` → `T[1]`

There are 8 chains of four, 3 chains of two, and two chains of one (`T[0]`→`T[0]` and `T[20]`→`T[20]`). Since PA-RISC can issue two memory operations per cycle but can retire only one store per cycle, the optimal ordering loads from one chain, then interleaves the stores with the loads from the next chain. The expected performance is 40 cycles, which is the number of times a multiple cycle loop would have to run to perform the same task. It also eliminates the need for a temporary key array: the target key array can be used for all intermediate values.

IA-64

The IA-64 Mars keying implementation takes advantage of the large register files, rotating registers, and rotating predicates. The routine allocates a 48-register stack frame, all of which are rotating. The initial register usage is as follows:

r32-r34	r35	r36	r37	r38	r39	r40	r41	r42	r43	r44	r45	r46	r47	r48	r49	r50-r79
Unused	k_X	k_3	k_2	k_1	k_0	T_i	T_{i-1}	T_{i-2}	T_{i-3}	T_{i-4}	T_{i-5}	T_{i-6}	T_{i-7}	A	B	T[10..39]

The first nine computations simply initialize `T`$_i$ and rotate registers to the right. After that, the registers `A` and `B` contain the first two values for stirring. Unlike PA-RISC, this phase of the computation is enabled by the rotating predicates, where a '1' is shifted in each time through the main body of the loop. To circulate the key words, $k_0 \rightarrow k_X$ at the end of the loop. When the initialization phase of the loop is finished, the loop switches to the epilogue phase, which now shifts a '0' into the rotating predicates, which disables the initialization instructions. Thus, the entire expansion/mix phase executes in one loop that runs 48 times, 6 cycles per loop.

When the first phase is finished, the intermediate key values are in the rotating registers, with `r39` = `T[0]`, `r38` = `T[1]`, …, `r32` = `T[7]`, `r79` = `T[8]`, …, `r48` = `T[39]`. This allows the stirring phases to compute on the rotating register file. Since the registers rotate 39 places during the stirring loop, the registers used in each phase are:

Pass	T[i]	T[i-1]	T[0](Final)
2	r39	r38	r78
3	r78	r77	r69
4	r69	r68	r60
5	r60	r59	r51
6	r51	r50	r42
7	r42	r41	r33

The reorder is efficiently handled in a two cycle loop. In the first cycle, the key word is stored, the data pointer incremented seven words, and a look-ahead target index counter is tested for overflow and incremented. In the second cycle, the index and data pointers are adjusted if the index had overflowed in the previous cycle.

A comparison of AES candidates on the Alpha 21264

Richard Weiss
VSSAD Labs
Compaq Computer Corp,
334 South St
Shrewsbury, MA 01545
Richard.Weiss@Compaq.com

Nathan Binkert
Computer Science Dept
University of Michigan
Ann Arbor, MI
binkertn@umich.edu

ABSTRACT

We compare the five candidates for the Advanced Encryption Standard based on their performance on the Alpha 21264, a 64-bit superscalar processor. There are several new features of the 21264 that have a significant impact on encryption/decryption speed. The main ones are greater potential for instruction-level parallelism (ILP) and larger level 1 cache. The ILP comes from the fact that the 21264 can issue four integer instructions per cycle. We envision that for high-performance servers, there will be multiple streams of data for encryption or decryption. The type of parallelism that we consider in this paper is the encryption of multiple, independent blocks interleaved in the same code loop running on the *same processor*. This benefits some algorithms more than others. Rijndael and Twofish turn out to be the fastest for a single block at a time, but RC6 is potentially the fastest when processing two blocks at a time. The reason for this is that out-of-order execution together with an issue width of four can be used to hide the latency of integer multiplies.

Introduction

The new AES algorithms will be used on a wide range of CPU's. The Alpha 21264 is a good representative of a 64-bit RISC architecture. Its features include a 64K two-way set associative level-1 cache, the capability to issue 4 integer instructions each cycle, and out-of-order execution. Since the Alpha is most likely to be used in servers, it will probably be used for encrypting or decrypting multiple streams of data simultaneously. This can be done on multiple processors, but it is also relevant to look at the efficiency of processing more than one block simultaneously on each processor, thus increasing the throughput of the system. In the remainder of this paper, we will use the term ***multiple stream*** or ***multistream*** to refer to more than one block on the same processor. Most of the studies so far have looked at single stream performance, where latency is the dominant factor. In order to get optimal multistream performance, it will be necessary to harness the full bandwidth of the processor. The five candidate AES algorithms have different computational requirements, and therefore have different behavior with respect to multistream than single stream.

We illustrate the multiple stream scenario with an example, so that there is no ambiguity. Consider the following assembly language fragment from a loop for an

imaginary processor that can issue two instructions per cycle, at most one of which can be a multiply:

```
loop:
      1. Load S[0]         # load key
      2. T = Mull A*A

      3. Load S[1]         # load key
      4. U = Mull B*B

      5. C = Shift_right  T
      6. D = Shift_left   T

      7. E = Shift_right  U
      8. F = Shift_left   U

      9. C = C Or D
      10. E = E Or F

      11. B = C Add S[0]
      12. A = E Add S[1]

      13. Br loop
```

The processor will execute two instructions per cycle except for the branch. If the latency of each instruction were one cycle, then the whole code would take seven cycles. However, if the latency of a multiply is seven cycles and at most one can be issued in a given cycle, then there is a five cycle stall after the fourth instruction. Therefore, the execution time increases to 12. Now consider what we can do for two independent blocks of data:

```
loop:
      Load S1[0]         # load key1
      T1 = Mull A1*A1

      Load S1[1]         # load key1
      U1 = Mull B1*B1

                         C2 = Shift_right  T2
                         D2 = Shift_left   T2

                         E2 = Shift_right  U2
                         F2 = Shift_left   U2

                         C2 = C2 Or D2
                         E2 = E2 Or F2

                         B2 = C2 Add S2[0]
                         A2 = E2 Add S2[1]

                         Load S2[0]         # load key2
                         T2 = Mull A2*A2

                         Load S2[1]         # load key2
                         U2 = Mull B2*B2

      C1 = Shift_right  T1
```

```
D1 = Shift_left  T1

E1 = Shift_right  U1
F1 = Shift_left   U1

C1 = C1 Or D1
E1 = E1 Or F1

B1 = C1 Add S1[0]
A1 = E1 Add S1[1]

Br loop
```

The combined loop can process two blocks in only 13 cycles. The processing of the two blocks can be overlapped in such a way that while the shift operations for one block are waiting for the multiplies to complete, operations on the other block can proceed. For the 21264, the latency for a multiply is actually seven, and the latency of a load is three or more, depending on whether or not the value is in the D-cache. The 21264 can issue up to four integer instructions in one cycle, at most two of which can be loads. The out-of-order processing capability is not actually used if the compiler schedules the instructions to take into account the latency. It should be noted that future generations of Alpha processors will have simultaneous multithreading (SMT), which will eliminate the necessity of the programmer/compiler merging two streams of data in one instruction stream.

The key to taking advantage of the full issue width of the Alpha is recognizing when a program has a low number of instructions per cycle (ipc). In the above example, this was caused by the long latency of the multiplies, but there may be other cases where this happens. For example, in the implementation of Serpent that we used, there were long chains of dependent logical operations, which resulted in an ipc of slightly less than two. Thus, Serpent can achieve a speedup of almost two by processing two streams. RC6 is similar to the example above in that the multiplies introduce latency, which reduces the ipc to a level for which processing two streams works well. On the other hand, Rijndael, Twofish and Mars do not lend themselves to this approach. They can be coded efficiently for single stream so that the table lookups can be overlapped with the other computation and the ipc is well over two. It should be noted that an ipc of greater than two does not preclude multistream processing, but the gains are likely to be small. Also, it is important to use an optimized version of the code, otherwise a low ipc will only reflect the inefficiency of the implementation rather than the potential for multistream parallelism. For this reason, we examine assembly language implementations in addition to the C versions.

One of the architectural features that is missing from Alpha is the 32-bit rotate. This requires several instructions to emulate. A fixed rotation requires two shifts an “and” and an “or”. These can be executed in two parallel chains and in the absence of other parallelism they have an ipc of two.

The next section presents an analysis of each algorithm in terms of ipc for a C implementation and for an assembly code implementation.

Analysis of Algorithms

Our goal is to get a quick estimate of the performance for multistream data. We do this by checking the timings for the Gladman C implementations of the five candidate algorithms for single stream data and estimating the ipc. Then in some

cases, we also look at assembly language implementations to see if the ipc could be increased. While a high ipc will rule out a gain from multistream, a low ipc does not guarantee one. A range of techniques was used from a complete implementation in assembly language in the case of Rijndael, to coding a single round in assembly language for Rc6 and Twofish, to a data dependency anlysis for Mars and Serpent. The data dependency analysis together with instruction latency was used to estimate optimal times for the last two algorithms. In the one case where we did an assembly language implementation, the time for this was compared with our estimate. Finally, we estimated the gains for multiple stream implementations.

Mars

The Mars algorithm has three phases: simple arithmetic and logical operations, table lookup and rotations. The table lookup, which is mixed with some fixed rotations has a four-fold parallelism. This seems to be the reason for a high ipc, and therefore little gain from multistream. Since the Alpha does not have a 32-bit rotate, this increases the number of instructions. For this reason, it is both one of the fastest algorithms on a Pentium Pro but one of the slowest on the 21264.

RC6

RC6 turns out to be a lot more efficient on the Alpha 21264 than expected from observing the number of cycles for a single block of data. For single stream performance, each round when coded in assembly language, takes 18 cycles and there are 20 rounds. If we allow 20 cycles for setup, this gives a total of 380 cycles per block. This is amazingly close to the current reported figure of 382 cycles per block for the optimized C version. A single round of encryption for two independent blocks of data simultaneously was also coded in assembly language for an estimated 21 cycles, which is less than 11 cycles/block. For 20 rounds, this would be 210 cycles/block plus the time for setup and storing results. This is as fast as Rijndael, and is potentially more consistent since it uses multiplication, which have a fixed latency, and does not depend on table lookups which could suffer occasional cache misses. In addition, if the algorithm were used with a word size of 64, this could potentially double the throughput, since the 64-bit versions of the operations multiply, xor, add and rotate are as fast or faster than the 32-bit versions on Alpha processors.

Rijndael

The simplicity of the Rijndael algorithm makes it easy to analyze. We were able to produce an efficient implementation in assembly code together with timing results. The major computational cost for this algorithm is accessing the look-up tables. This can be done in three instructions: extract byte, add to base address, and load the value. For Alpha, this is relatively fast, since the tables fit in the level-one cache. Ideally, one round of Rijndael could be done in 18 cycles: however, in practice, this requires tuning the code to eliminate I-cache misses, D-cache misses, etc. What we observed was that the code took 246 cycles/block when executed repeatedly. This is about 23 cycles per round. This was the fastest algorithm we have observed for 128-bit key length. However, since the number of rounds for Rijndael depends on the key length, this is not the fastest for all applications.

We expect the Rijndael algorithm to scale well with future processors since the makeup of the code is such that one quarter of the instructions are loads. The Alpha 21264 can issue four integer instructions per cycle, and there is a four-fold parallelism from the four S-boxes. However, this gives it a high ipc and means that there is little gain from multistreaming. A single round of

Rijndael takes 18 cycles. The setup and exit code adds another 30 cycles to the total to give approximately 210 cycles per block.

Serpent
Based on the C-code from Brian Gladman, this algorithm is the slowest. However, it speeds up very well with multistreaming. The S-boxes are implemented by sequences of bit-parallel logical operations. Due to data dependencies in this code, the ipc is slightly less than two. The technique for estimating the two stream performance was to modify the C code. Each round is composed of three macros: an “xor” with the key, an S-box computation, and a linear transform. The processing of the two streams was interleaved by repeating each macro for the first stream with the identical macro for the second stream. The compiler was able further mix the instructions to eliminate stalls. Nevertheless, Serpent remains one of the slower algorithms because of the large number of rounds and the large number of instructions per round. It should be noted that most of the operations in Serpent operate on bits in parallel. It should be possible to process two blocks of 32-bit words by using the full 64-bit data path. Namely, one block would use the upper 32 bits, and the other block would use the lower bits. There would be an extra “and” for the rotates as well as packing the two words together, but the speedup could be close to 2x.

Twofish
Based on an assembly language coding of a single round, twofish performs approximately as well as Rijndael on both the 21164 and the 21264 for 128-bit key length. Since Twofish does not require more rounds for larger key lengths, its relative performance would be better for longer keys. It can potentially do eight S-box lookups in parallel for each round. This gives it a high ipc and small gain for multistreaming.

Timing Results

Table 1 shows the results from optimized C-code for the Alpha 21164 and 21264 processing one block at a time. The 21164 can issue two integer instructions per cycle and the 21264 can issue four. The results are similar to those published by Granboulan [Gran]. Our timings were all obtained by running each of the algorithms for key setup, encryption and decryption on a single stream of data, one block at a time. The C-versions of these algorithms are the ones published by Gladman [Glad1]. We ported them to Alpha by using the native cycle count register and modifying the declarations to eliminate alignment errors in the code. The basic idea is to time the execution of the encryption (decryption) code running once, then time it running twice. The minimum times over a large number of iterations are subtracted to measure the time to execute the code without the startup costs. In addition, the encryption (decryption) code is run once at the beginning to warm up the caches.

In order to relate our assembly code estimates to the C implementations, we linked our assembly version of Rijndael to the Gladman harness and observed an encryption time of 280 cycles/block. The assembly code when executed for a large number of iterations took a minimum of 246 cycles/block. This suggests that the C++ overhead for calling some of the C or assembly functions could be significant.

In Table 2, we have estimated timing results for assembly language implementations for some of the algorithms for single stream. Table 3 shows the estimated timing for assembly code for processing multiple streams.

EV56 (21164)	Mars	RC6	Rijndael	Serpent	Twofish
Ours	701c	571c	439c	984c	442c
Granboulan website	507c	559c	490c	998c	490c

EV6 (21264)	Mars	RC6	Rijndael	Serpent	Twofish
Ours	515c	428c	293c	854c	316c
Granboulan website	450c	382c	285c	855c	315c

Table 1. Timing comparison in cycles/block for C code.

EV6 (21264)	Mars	RC6	Rijndael	Serpent	Twofish
Assembly code	375c	360c	210c	570c	255c

Table 2. Estimated timing for assembly code in cycles/block.

EV6 (21264)	Mars	RC6	Rijndael	Serpent	Twofish
Assembly code	375c	210c	210c	506c	255c

Table 3. Estimated time for assembly code encrypting two blocks simultaneously. Times are in cycles/block.

Conclusions

RC6 has the most potential for parallelism when multiple streams are processed on the same processor simultaneously in a single thread. One reason for this is that it relies heavily on multiplication, which itself has a large degree of parallelism for the Alpha processors. 32-bit multiplies are inherently parallel because they operate on four bytes at the same time. Using 64-bit multiplication would afford even more parallelism. The 21264 can issue one multiply every cycle. The latency of seven cycles does not limit bandwidth for this algorithm in multistream mode. An S-box lookup requires three instructions, and only operates on one byte at a time. Note that while RC6 has variable 32-bit rotations, one of the intermediate results from the fixed rotation by 5 is re-used in the variable rotation.

Serpent also has a large gain from multistream processing because of the long dependent chains of instructions and low ipc. However, because of the large number of rounds and instructions per round, it still is slow.

Following RC6 are Twofish and Rijndael, which both use 8-bit table lookups and linear transforms. Twofish has an advantage for longer keys, but Rijndael seems the fastest for 128-bit keys. Based on an assembly language implementation of

Rijndael, there can be a significant difference between the estimated performance and what can be readily achieved/observed by counting cycles outside of the algorithm function call. Comparing code execution with timing estimations can have a significant amount of error.

Since our estimates for the Alpha 21264 are based on instruction level parallelism for processing multiple streams, similar behavior should be observable for Itanium and other VLIW machines.

Acknowledgements.
We would like to thank Dr. Brian Gladman for publishing unified C implementations of the five AES candidate algorithms. Also we thank Steve Root for assembly language implementations of some of the algorithms.

References

[KA] Almquist, Kenneth. "AES Candidate performance on the Alpha 21164. http://home.cyber.ee/helger/aes/kenneth.txt

[Glad1] Gladman, Brian. "Implementation experience with AES candidate algorithms." Second AES Conference, Feb, 1999. http://jya.com/bg/gladman.pdf

[Glad2] Gladman, Brian. http://www.btinternet.com/~brian.gladman/cryptography_technology/Aes/index.htm

[Gran] Granboulan, Louis. "AES Timings of best known implementations." http://www.dmi.ens.fr/~granboul/recherche/AES/timings.html

[SKW] Schneier, B., Kelsey, J., Whiting, D., et al. "Performance Comparison of the AES Submissions."

Performance Evaluation of AES Finalists on the High-End Smart Card

Fumihiko Sano* Masanobu Koike* Shinichi Kawamura† Masue Shiba*

* Toshiba System Integration Technology Center
3-22, Katamachi Fuchu-shi, Tokyo, 183-8512, JAPAN
† Toshiba Research and Development Center
1, Komukai Toshiba-cho, Saiwai-ku, Kawasaki, 210-8582, JAPAN
{fumihiko.sano, masanobu2.koike, shinichi2.kawamura, masue.shiba}
@toshiba.co.jp

Abstract. This paper reports the performance of the AES finalists, MARS, RC6, Rijndael, Serpent, and Twofish, on the high-end smart card that has a Z80 core with Toshiba's arithmetic coprocessor.

1 Introduction

During the first round of AES candidate assessment, some reported the performance evaluation of the algorithms on low-end smart cards. Their reports are important for understanding performance of each AES candidates in memory and computing resource-restricted environments. However, there are, so called high-end smart cards, which are equipped with a specific hardware for accelerating cryptographic processing. In general these cards are less restricted in their resource than low-end smart cards. So, it is important for better understanding of the AES candidates to evaluate the performance on high-end smart cards. NIST as well expressed their interests in such evaluation in [11]. This paper describes our experience in implementing five AES finalists, and summarizes the performances on our high-end smart card available from Toshiba[17].

The high-end smart card is substantially different from low-end one in that its core is equipped with a crypto coprocessor. It may usually correct to say that the amount of memory for a high-end card is larger than that of low-end one. In some cases, however, venders supply cards with large memory amount suitable for their specific purposes regardless of the core. Therefore, we distinguish between high-end and low-end cards based on the type of core.

At first, we present the architectures of the core on our smart card that includes a CPU and a coprocessor architecture. Next, we describe coding rules for our implementation and then, present experiences of five AES finalists accompanied by results of 64-bit ciphers such that DES[10] and MISTY1[9] on our smart card for reference purpose. Finally, we summarize advantages and disadvantages for each implementation.

2 Platform

High-end smart cards available now are usually equipped with 8/16-bit microprocessor and a crypto coprocessor, or accelerator for cryptographic operations[7]. To evaluate the AES finalists' performance on high-end smart cards, we choose Toshiba's T6N55 chip shown in table 1. The chip is equipped with Z80 microprocessor and a coprocessor. The coprocessor is under the control of Z80 and it carries out arithmetic/logical operations when Z80 asks to do so. The coprocessor is originally designed to accelerate the large integer arithmetics. As will described shortly, it is also suitable to accelerate some operations required to implement AES finalists.

Table 1. Features of Toshiba's T6N55 chip

CPU	Z80
ROM	48KB
RAM	1KB
EEPROM	8KB
Max. of Modulus	2,048-bit
Internal Clock Frequency	5MHz

2.1 Z80 Architecture

The Z80 is a famous 8-bit architectured microprocessor developed by ZiLOG[15]. It has an 8-bit accumulator and a flag register, six 8-bit general-purpose registers, two 16-bit index registers, a stack pointer (SP), and a program counter (PC). An accumulator A and a flag register F can be paired and dealt with as if it is a 16-bit register AF. Similarly, 8-bit registers can be paired with particular registers as BC, DE, and HL. Z80 incorporates dual register banks. Each register bank has each register sets such as an accumulator, a flag register, and six 8-bit registers. Note that one can use only one side of the banks at a time. If one want to use registers belonging to the other side of the bank, he should change the contexts with an EXX operation.

The instruction set includes the following classes:

- Load 8-bit values to registers or an accumulator.
- Load 16-bit values to registers.
- Arithmetic or logical instructions for the accumulator with registers.
- A single bit shift or rotate instructions.
- Compare, block transfer, and search instructions.
- Branch instructions.
- Subroutine calls and returns from them.
- I/O instructions.

– Checking or setting a single bit in registers.

There are some particular instructions for extended registers or control instructions of processor. Z80 can execute addition, subtraction, AND, OR, exclusive or (XOR), and single-bit rotation and shift. It does not have instructions for multiplication and division.

On using the ordinary Z80 core, we should take some features of its architecture into account. It needs four clocks even for the basic instructions, such as a no operation (NOP) or a load instructions between registers (LD r, r'). The next fastest instructions, such as for loading a value to a register (LD r, n) consume seven clocks. Operations for 16-bit register sets are more time consuming. Although we try to use faster operations, the average number of clocks needed for an instruction is about six.

2.2 Crypto Coprocessor

The coprocessor is developed mainly to accelerate the processing of the public key cryptosystem. It has 512-byte RAM area (we call it the 'CRAM' area). That area is segregated into two 256-byte RAM areas. The coprocessor can execute various operations between the 256-byte RAM areas or on the 512-byte RAM. Each maximum size of arithmetical operations supported by the crypto coprocessor is shown in table 2.

It can execute the following classes of calculations:

– Addition, subtraction, multiplication, division, and logical operations.
– Modular multiplication.
– Modular exponentiation.
– Montgomery multiplication.
– Extended Euclidean algorithm.
– Memory transfer in CRAM area.

Here, the logical operations mean AND, OR, and exclusive OR(XOR). The memory transfer is used to transfer data on CRAM area efficiently. So, the feature is similar to the direct memory access (DMA). The most time consuming operation is a modular exponentiation with a large exponent. Other operations, when used in implementing AES finalists, are very fast and finish within a time for the minimum execution time of a Z80 instruction.

The coprocessor executes logical operations between operands located on each CRAM areas. Before executing these operations, Z80 have to put several bytes of control words on the CRAM area in addition to the operands. Since Z80 does not perform so fast to the data on memory, using coprocessor operations are efficient for large data, but not so much for small data.

3 Implementations

3.1 Coding Rules

When we implement the AES finalist, we apply the following rules for the coding.

Table 2. Features of Toshiba's Crypto Coprocessor

Instruction	Max. of Operands (bits)
Addition	2,048
Subtraction	2,048
Multiplication	1,024
Division	2,048
Modular Multiplication	1,024
Exponentiation	1,024

- Program codes are located on the ROM area, and we do not change the code at any time.
- We can use all registers, i.e., registers on both sides of the banks.
- The codes run in constant time not depend on the data to avoid timing analysis.
- We can use memory on the CRAM area if necessary.
- We write codes that generate the extension keys with on-the-fly, if possible.

A time constancy of a code is an imprecise term. We try to give more precise idea behind the third rule. If we have only to realize the time-constancy, we may choose an easy way to stretch the execution time by merely adding NOPs at the end of the code. But what we really have to do is to avoid timing analysis. So, we have to pay more attention not to leak meaningful information. If we can successfully apply the third rule, we can prevent simple power analysis as well as timing attack. The third rule is not sufficient, though it seems necessary, to prevent the differential power analysis. We don't discuss on the differential power analysis in this paper any further.

It is interesting that we may neglect the differences between rounds, for example the key expansion of DES need 2-bit rotations in some rounds. They may leak some information, but it seems useless for analysis.

In this section, we report the performance of AES finalists in alphabetic order. For comparison purpose, results for 64-bit block ciphers, such as DES, triple DES, and MISTY1, will be shown, as well. We describe the speed of each algorithm with clocks and RAM requirement: In each table, 'Int.' means that size of required CRAM for coprocessor's operations, and 'Ext.' means other work area. Note that 5,000 clocks at 5MHz correspond to 1 millisecond. For example, DES needs about 25,000 clocks, and thus it works in 5ms.

The code of DES does not necessarily obey the coding rules above since some permutations for DES are realized by hard wired logic. The triple DES is a two-keyed one, but it executes the key schedule three times with on-the-fly. Therefore, three-keyed triple DES will have the same performance result. MISTY means the MISTY1 algorithm[9] with eight rounds.

To apply our results easily for other processors that have similar features, we try to reduce the memory usage on the CRAM area. But, in this paper, we see that the memory usage is of little importance, since the platform chosen has sufficient memory for these implementations.

3.2 MARS

It is the most difficult task for us to implement MARS on smart cards or other limited resources. MARS has a complex high level structure such as eight rounds of unkeyed forward mixing, eight rounds of keyed forward transformation, eight rounds of keyed backward transformation, and eight rounds of unkeyed backward mixing. Each of the eight rounds consists of so called type-3 Feistel network. In a type-3 Feistel network, input data is segregated into four words. One of them is taken as a pseudo-random function's input and the output is used to modify three other data words. Since MARS has a block length of 128 bits, each word has 32 bit length.

There are three disadvantages of MARS when implemented on a smart card. The first is that it needs 2KB table for S-boxes, but it is not serious. The second is the weakness check of extended key on the key schedule. The last is the rotations with variable shift amount. We discuss the last two disadvantages here.

It is necessary for MARS to implement complicated "weak" measures on the key schedule[3]. The weak keys for MARS are different from those of DES. In the case of DES, you may disregard the problem of weak key because it only increases some potential threats caused by the weak key properties. However, in the case of MARS, since the weak key check procedure is a part of the algorithm specification, you have to check the weak on the key schedule certainly. Otherwise, you may see a terrible result, such as differences in cipher text, although it encrypts the same plain text with common key. As mentioned above, the function of checking the weak on the key schedule is primarily needed.

Although implementing weak key check is necessary, it is also true that this introduces another problem for smart card implementation. If we check the weak and regenerate extension keys, there is a risk of applying timing attack. The regeneration of extension keys causes difference in processing time and leaks some information on the key. Further study of coding is necessary to avoid this problem.

To save our time, our implementation just omits the weak key check. Therefore, it is not complete. Our implementation is not so slow because of customization for 256-bit key and omitting to check 'weak' on the key schedule. The codes for check 'weak' on the key schedule will increase the requirement of ROM and processing time.

The rotations depend on a key data or an internal data are crucial for Z80 or other 8-bit processors since we need to write codes that run in constant time, or else an attacker can get some information about the key. Fortunately, our coprocessor can operate modular multiplications over any modulus. We use them for rotations. Modular multiplications on our smart card are very fast, and finish within a single instruction of Z80. It means that we can operate modular multiplications and data dependent rotations in a constant time and avoid timing attack.

It seems that MARS is a prudent algorithm against cryptanalysis. But it causes some difficulties in implementing on smart cards or similar resource-restricted environments.

Table 3. MARS

	RAM (bytes)			ROM (bytes)	Time (clock)
	Total	Int	Ext		
Encrypt	60	36	24	3,977	45,588
Schedule	512	512	0	1,491	21,742
Total	512	512	24	5,468	67,330

3.3 RC6

RC6 has various parameters and is defined as RC6-$w/r/b$ where w means the word length, r means the number of rounds, and b means the length of key with bytes. We write the code with the recommended parameters for AES such as RC6-32/20/32.

RC6 has a simple structure, but the round function includes various operations such as, addition, subtraction, multiplication, and rotations depending on a variable data. Most part of RC6 constructed by arithmetical operation. Therefore, we operate almost all operations on the coprocessor. Furthermore, since the coprocessor can operate up to 1,024 bits for operand, we can execute the pair of rotations with constant shift amount in parallel. An n-bit rotations to two data is written as follows: We duplicate each of data and put them on corresponding CRAM area, then multiply them with 2^n. As a result, we can improve the performance and reduce the size of code.

The coprocessor can execute RC6 data encryption efficiently. RC6 has a simple key schedule but need much iterations and does not suitable with on-the-fly. The key schedule takes four times as long execution time as encryption.

There is an idea to improve the key schedule processing time. A precomputed table improves the speed, but increase the size of code. It omits the computation of 43 initial values (S[i]) with 32-bit word. The modified code copies S[i]s from precomputed ROM table to RAM area instead of computing S[i]s with constant values. It shall reduce about 4,000 clocks. It needs some extra code or table for precomputed table, thus the size of code increases about 150 bytes.

On the smart cards, RC6 has a moderate encryption speed among the finalists, but its key schedule is slower than Rijindael or Twofish. Note that it has been reported that on the 32-bit processor, RC6's performance is faster than Rijndael and Twofish[5].

Table 4. RC6

	RAM (bytes)			ROM (bytes)	Time (clock)
	Total	Int	Ext		
Encrypt	124	124	0	489	34,736
Schedule	90	90	0	571	138,851
Total	156	156	0	1,060	173,587

3.4 Rijndael

256-bit key is the fastest for on-the-fly key generation, since we can translate the internal key every two rounds. 128-bit key is a little slower than 256-bit key, since we need to make extension keys every round. In the case of 192-bit key, since the key length is not the multiple of the block length, it is not so easy to implement on-the-fly key generation.

The `xtime` is an important subroutine for time constancy. It needs modulus operation with the primitive polynomial. Here is an example of straightforward implementation of the `xtime(a)` algorithm where the original value is stored in A register.

```
        RLA
        JR    NC, SKIP
        AND   PRI       ; PRI means the primitive polynomial.
SKIP:
        ...             ; end.
```

This is a very dangerous code. Since 'AND *PRI*' operation is operated only when the carry is '1', an attacker can know whether the value excesses 2^8 or not in this code. We must avoid such an implementation. Therefore, we use some techniques to avoid differences of processing time and thus prevent cryptanalysis using timing attack. Here is an example of `xtime(a)` operation with constant time, where `a` is stored in A register.

```
        RLA
        LD      B, A
        SBC     A, A
        AND     PRI
        XOR     B
```

RLA is a instruction of 1-bit leftward rotation for A register. If RLA is carried out, MSB of A register is set to the carry flag. 'SBC A, A' is an instruction which substract a value in A register and a carry from A register. It means that if the carry flag is '1' then A register has a value `0xff`, otherwise A register has a value `0x00`. Next we operate AND instruction with *PRI* for A register. Then we get *PRI* or a value `0x00` in A register, and we can operate whether 'XOR PRI' or 'NOP' with the same instructions and processing time.

The transformation MixColumn is implemented in an efficient way shown in section 5.1 in [4]. We implement the `AddRoundKey` and data transfers with the coprocessor. Other transformations in Rijndael are not so heavy even for only the Z80 core. Rijndael is the most efficient algorithm on the finalists on our smart card.

A disadvantage of Rijndael is that it needs another code for decryption because of the asymmetry of encryption and decryption. If you need both encryption and decryption algorithms, it takes twice ROM area for code since most part of it cannot be shared.

Table 5. Rijndael

	RAM (bytes)			ROM (bytes)	Time (clocks)
	Total	Int	Ext		
Encrypt	34	32	2	700	25,494
Schedule	32	32	0	280	10,318
Total	66	64	2	980	35,812

3.5 Serpent

There is two kinds of implementation of Serpent: ordinary implementation and bitsliced implementation. Here is the result of an ordinary implementation of Serpent. It is not a bitsliced implementation. It needs a 2,048-byte ROM table on the ordinary implementation.

Serpent has various rotational operations. As is described in MARS implementation, modular multiplication with coprocessor can be used if they improve the performance. Most of the rotations are, however, more efficient with the Z80 operations than with the coprocessor. 1-bit leftward or rightward rotations can be implemented with the Z80 operations, and shifts with multiplies of 8-bit are reorder of bytes. We use the coprocessor operations only for 11-bit rotations, XOR, and memory transfer. Due to the architecture of our coprocessor, it is not suitable to efficiently implement three-operand operation used in Serpent.

In [2], Serpent can be implemented using under 80 bytes of RAM with on-the-fly. Our implementation needs twice more RAM, because we write it with coprocessor's operation XOR between halves of CRAM with different offsets.

It has more rounds than other finalists do, so its performance is not so good as Rijndael or Twofish.

The bitsliced implementation will reduce the size of code and required RAM with a little degradation in speed. In memory-restricted environment, bitsliced implementation may be better than the ordinary coding. In this paper, we attach importance to the speed. So, we choose the ordinary implementation for performance comparison.

3.6 Twofish

In the case that the length of key is less than 256-bit, we need to pad out the original key until it becomes 256-bit. We implement Twofish with 128-bit key to

Table 6. Serpent

	RAM (bytes)			ROM (bytes)	Time (clock)
	Total	Int	Ext		
Encrypt	68	68	0	3,524	71,924
Schedule	96	96	0	413	147,972
Total	164	164	0	3,937	219,896

take the processing time for padding into account. It includes code for padding, and it is a little slower than 256-bit key.

There are two models for implementing Twofish, such as Feistel model and non Feistel model[14]. We implement it with non Feistel model. We assume that it is faster than Feistel. We use coprocessor's operations for additions with subkeys, XOR, and memory transfers on CRAM area, but rotations are implemented with Z80's rotations.

The performance of Twofish depends on the size of precomputed tables' [14]. We consider that the case of using some tables amounted to 1,536 bytes. This code is compact for processing the key schedule with precomputed tables. It seems be compatible with 2200 bytes for code and table size model in [14]. The size of precomputed tables is belongs to encryption code in table 7.

Twofish is as fast as DES on throughput. It does not have any exceptional advantages, but we have nothing to complain about the performance.

Table 7. Twofish

	RAM (bytes)			ROM (bytes)	Time (clock)
	Total	Int	Ext		
Encrypt	34	32	2	2,493	31,877
Schedule	56	32	24	315	28,512
Total	90	64	26	2,808	60,389

4 Summary

We summarize the performance and the required resources on our implementations in table 8. The RAM includes required byte in the RAM area and the CRAM area. Note that when using a coprocessor, the required amount of RAM increase, because of the alignment rules for CRAM area.

Some finalists are designed to have heavy key schedules. They are intended to prevent exhaustive search attacks, but resulting in speed reduction on smart cards. We consider that Rijndael is excellent on all aspects. RC6 is as good as Rijndael on the code point of view, but the key schedule consumes more time.

Twofish needs much ROM memory than RC6 and Rijndael because of the table. It is faster than Triple DES and equal to DES on the throughput. It will have good performance on any smart cards. MARS has disadvantages of its code size caused by four of eight round iterations and a 2,048-byte table. The speed is equal to Twofish's one. We consider MARS has some difficulties to check 'weak' on the key schedule and regenerate. Serpent has disadvantages of its performance caused by the iterations of rounds and the difficulty of key schedule. The bitsliced implementation will improve the requirement of ROM or RAM, but slower than others.

We tried to write all program codes to consume as little RAM area as possible. On the other hand, if we may regard the RAM area, especially CRAM area, as a kind of free work space, it will be unfair to compare finalists how little work area they consume. Nevertheless, notice that MARS consumes all the CRAM area, whereas others consume at most half of the area.

Table 8. Comparison of AES finalists and the algorithms

Cipher	RAM (bytes)		ROM (bytes)		Time (clock) Encrypt		Schedule		Encrypt + Schedule		
MARS	572	5	5,468		45,588	4	21,742	2	67,330	3	*
RC6	156	3	1,060	2	34,736	3	138,851	4	173,587	4	
Rijndael	66	1	980	1	25,494	1	10,318	1	35,812	1	only encryption
Serpent	164	4	3,937	4	71,924	5	147,972	5	219,896	5	
Twofish	90	2	2,808	3	31,877	2	28,512	3	60,389	2	
DES	17		772						25,398		
Triple DES	17		849						72,341		
MISTY	44		1,598						25,486		

$*$: omit to check "weak" in the key schedule.

5 Conclusion

We have implemented AES finalists on a high-end smart card that is equipped with a crypto-coprocessor. The resulting code has higher performance than that on a low-end smart cards, since multiplication and rotation are efficiently implemented using the coprocessor's commands. Coprocessor's RAM are also useful for work memory, as well.

Regarding speed, Rijndael is the best one and is as fast as our DES implementation. It is twice faster than DES on the throughput. RC6 is suitable for our smart card same as on the 8051[6, 8], but not to be compared with Rijndael or Twofish because of the key schedule.

For smart card implementation, it is necessary to perform key schedule at least for every processing block, in order to save memory areas to store extended

key. For the same reason, it is desirable for key schedule to be suitable for on-the-fly key generation. As a result, design concept for key schedule affects the performance very much, and those algorithms that have heavy key schedule are not advantageous for smart card implementation.

Finally, we report the performance of E2[12] that is a candidate on the first round in the appendix.

References

1. R. Anderson, E. Biham, and L. Knudsen, *"Serpent: A Proposal for the Advanced Encryption Standard"*, AES submission, 1998.
2. R. Anderson, E. Biham, and L. Knudsen, *"Serpent and Smartcards"*, CARDIS '98, 1999, available on http://www.cl.cam.ac.uk/~rja14/serpent.html.
3. C. Burwick, D. Coppersmith, E. D'Avignon, R. Gennaro, S. Halevi, C. Jutla, S. M. Matyas Jr., L. O'Connor, M. Peyravian, *"MARS -a candidate cipher for AES"*, AES submission, 1998.
4. J.Daemen, V.Rijmen, *"AES Proposal: Rijndael"*, AES submission, 1998.
5. B. Gladman, "AES Algorithm Efficiency", http://www.btinternet.com/~brian.gladman/cryptography_technology/Aes/
6. G. Hachez, F. Koeune, and J. Quisquater, *"cAESar results: Implementation of Four AES Finalists on Two Smart Cards"*, The second AES conference, 1999, available on http://www.dice.ucl.ac.be/crypto/CAESAR/caesar.html.
7. H. Handschuh, and P. Paillier, *"Smart Card Crypto-Coprocessors for Public-Key Cryptography"*, CryptoBytes, Vol. 4, No. 1, RSA Laboratories, 1998.
8. G. Keating, *"Performance Analysis of AES candidates on the 6805 CPU core"*, The second AES conference, 1999, available on http://www.ozemail.com.au/~geoffk/aes-6805/.
9. M. Matsui, *"New Block Encryption Algorithm MISTY"*, Fast Software Encryption, 4th International Workshop Proceeding, LNCS **1267**, Springer-Verlag, 1997, pp.54-68.
10. National Bureau of Standards, *"Data Encryption Standard"*, U.S.Department of Commerce, FIPS 46-3, October 1999.
11. J. Nechvatal, E. Barker, D. Dodson, M. Dworkin, J. Foti, and E. Roback, *"Status Report on the First Round of the Development of the Advanced Encryption Standard"*, http://csrc.nist. gov/encryption/aes/round1/r1report.pdf
12. Nippon Telegraph and Telephone Corporation, *"Specification of E2 – a 128-bit Block Cipher"*, AES submission, 1998.
13. R.L. Rivest, M.J.B. Robshaw, R. Sidney, Y.L. Yin, *"The RC6 Block Cipher"*, AES submission, 1998.
14. B. Schneier, J. Kelsey, D. Whiting, D. Wagner, C. Hall, *"Twofish; A 128-Bit Block Cipher"*, AES submission, 1998.
15. ZiLOG, *"Z80 Microprocessor Products"*, available on http://www.zilog.com/products/z80.html
16. http://csrc.nist.gov/encryption/aes/round2/Round2WhitePaper.htm, 1999.
17. http://www.toshiba.co.jp/about/press/1999_02/pr_j0301.htm, (in Japanese).

A E2

E2 is not selected as a finalist for the second round review. But it has a good performance, especially encryption speed without key schedule. The serious disadvantages of E2 are that it has time consuming key schedule and can't execute it with on-the-fly. Fortunately, since the RAM usage fits on the half of CRAM area, we select a way to extend all round keys on the half of them, at first. In this case, E2 is efficient for encryption just like the report in [6]. The round function is designed as suitable for byte oriented operations. It is good for the Z80 architecture. It is, however, difficult for Z80 to execute multiplication on the IT and division on the FT. We use the coprocessor's commands for these operations. Those commands include XOR, memory transfer, multiplication, and inverse.

Table 9. E2

	RAM (byte)			ROM (byte)	clock
	Total	Int	Ext		
enc	26	24	2	1,519	17,018
key	548	512	36	296	79,358
Total	548	512	36	1,815	96,376

How Well Are High-End DSPs Suited for the AES Algorithms? *

AES Algorithms on the TMS320C6x DSP

Thomas J. Wollinger[1], Min Wang[2], Jorge Guajardo[1], Christof Paar[1]

[1]ECE Department
Worcester Polytechnic Institute
100 Institute Road
Worcester, MA 01609, USA
Email: {wolling, guajardo, christof} ece.wpi.edu

[2] Texas Instrument Inc.
12203 S.W. Freeway, MS 722
Stafford, TX 77477, USA
Email: minwang micro.ti.com

Abstract

The National Institute of Standards and Technology (NIST) has announced that one of the design criteria for the Advanced Encryption Standard (AES) algorithm was the ability to efficiently implement it in hardware and software. Digital Signal Processors (DSPs) are a highly attractive option for software implementations of the AES finalists since they perform certain arithmetic operations at high speeds, they are often smaller and more energy-efficient than general purpose processors, and they are commonly used for the rapidly growing market of embedded applications. In this contribution we investigate how well modern high-end DSPs are suited for the five final candidates chosen after the second AES conference. As a result of our work we will compare the optimized implementations of the algorithms on a state-of-the-art DSP.

Keywords: cryptography, DSP, block cipher, implementation

*This research was supported in part through a graduate fellowship by secunet Security Networks AG and a grant from the Texas Instrument University Research Program.

1 Introduction

The National Institute of Standards and Technology (NIST) has initiated a process to develop a Federal Information Processing Standard (FIPS) for the Advanced Encryption Standard (AES), specifying an encryption algorithm to replace the Data Encryption Standard (DES) which expired in 1998 [14]. NIST has solicited candidate algorithms for inclusion in AES, resulting in fifteen official candidate algorithms of which five have been selected as finalists. Unlike DES, which was designed specifically for hardware implementation, one of the design criteria for the AES candidate algorithms is that they can be efficiently implemented in both hardware and software. Thus, NIST has announced that both hardware and software performance measurements will be included in their efficiency testing. Several earlier DSP's contributions looked into the software implementation of the AES algorithms on various platforms [1]. However, there was only one publication dealing with the implementation of the candidate algorithms on a Digital Signal Processor (DSP) [9].

Digital Signal Processors are a distinct family of micro processors. In comparison to the more common general purpose processors such as those offered by, e.g., Intel and Motorola, DSPs allow for fast arithmetic, special instructions for signal processing applications, real-time capabilities, relatively lower power, and relatively lower price (obviously, those statements tend to over-generalize and should not be taken too literally). The main application areas of DSPs are embedded systems, such as wireless devices, cable and Digital Subscribe Line (DSL) modems, various consumer electronic devices, etc. With the predicted increase of embedded applications and pervasive computing, it is not unreasonable to expect that DSPs and DSP-like processors will become more commonplace. At the same time, it seems likely that many future embedded applications will need some form of encryption capability, for instance, for assuring privacy over wireless channels.

The questions that we try to address in this contribution are: How well are high-end DSPs suited for the implementation of the AES finalists? Can modern DSPs compete with general purpose computers in terms of speed?

In this paper, we focus on the implementation of the five AES finalists on a Texas Instruments TMS320C6000 DSP platform. In particular, the implementations are on a 200 MHz 'C62x/'C64x which performs up to 1600/8800 million instructions per second (MIPS) and provides thirty-two/sixty-four 32-bit registers and eight independent functional units.

2 Pre ious ork Cryptography on SPs

The field of implementing cryptographic algorithms on special platforms is very active. However, the research done on implementation of cryptographic schemes on a DSP is limited. There are a few papers that deal with public-key cryptography. There is one previous paper about the implementation of the AES candidates on a DSP. The papers [3, 7, 10] deal primarily with the implementation of public key algorithms on DSP processors. The main conclusion of these papers is that DSPs are a good choice for these algorithms due to the integer arithmetic capabilities of DSPs.

Reference [7] also describes the implementation of DES on a Motorola DSP 56000. It was found that the algorithm encrypts at roughly the same speed as a contemporary PC (20 MHz Intel 80386).

Karol Gorski [9] commented on the set of the AES Round 1 candidate algorithms, based on the timings obtained on the TI TMS320C541 DSP. Reference [9] used the C implementation by Brian Gladman, compiled with full compiler level optimizations. The resulting low speeds of the algorithms were due to the 'C54x 16 bit operations which are not ideal for most of the AES candidates. There was also no effort made to optimize the algorithms beyond those optimizations automatically performed by the C compiler.

3 Methodology

3 1 The Implementation of the Fi e A S Finalists

We implemented Mars, RC6, Rijndael, Serpent and Twofish on a TMS320C6201 DSP. RC6 was also implemented on the C64x DSP. As the basis of the implementations we used either the reference or optimized C code provided by the algorithm's authors, or the C code written by Brian Gladman [8].

It is important to point out the way we chose to code each algorithm, because they all offer several implementation options. In [6], the authors of Rijndael proposed a way of combining the different steps of the round transformation into a single set of table lookups. Each table has 256 4-byte word entries. Similarly, our Twofish implementation uses the "Full Keying" option as described in the specification [13]. Inother words we used 4 KByte tables which combine both the S-box lookups and the multiplication by the column of the MDS matrix. RC6 is a fully parameterized encryption algorithm [11]. The version of RC6 that we implemented is RC6-32/20/16. Mars was coded in the original version as stated in the algorithm specifications in [4], with 8, 16, and 8 rounds of "forward mixing", "main keyed transformation", and "backwards mixing", respectively. Finally, in [2] the authors described an efficient way to implement Serpent. Thus, we implemented the S-boxes as a sequence of logical operations which were applied to the four 32-bit input blocks.

3 2 Tools and ptimi ation ort

The source code was first compiled using the standard Texas Instruments C compiler (versions 3.0 and 4.0 alpha), utilizing the highest level of optimizations (level 3) available. For further information about the levels of optimization performed by the compiling tools, see [15, page 3 2 and 3 3].

After the implementation of the C code version, we optimized the encryption and decryption functions of the algorithms so that the compiler could further optimize it. In order to do so, we took advantage of the 32-bit data bus which is capable of loading 32-bit words at a time. We performed math operations with *ntrinsic unctions* to speed up the C code. *ntrinsic unctions* are similar to an additional mathematical Run-Time Support (RTS) library. They allow the C code to access hardware capabilities of the 'C6x devices while still following ANSI C coding practices. We also tried to use as many of the functional units in parallel as possible, e.g., by replacing constant

multiplication by shifts, by unrolling loops, or by preserving loops.

We further rewrote the encryption and decryption function for most algorithms in linear assembly to achieve performance improvements. Linear assembly is assembly code that has not been register-allocated and is unscheduled. The assembly optimizer assigns registers and uses loop optimization to turn linear assembly into highly parallel assembly. However, we did not program in pure assembly which is a very challenging and time consuming task on a complex processor such as the 'C6201, with eight independent functional units.

3 3 Parallel Processing Single- lock Mode s Multi- lock Mode

In addition to the optimizations described above, we implemented a second version of code in which data blocks can be processed in parallel. With parallel processing, the encryption and the decryption functions can operate on more than one block at a time using the same key. This allows better utilization of the DSP's functional units which leads to better performance.

With parallel processing, however, the speedups may only be exploited in modes of operations which do not require feedback of the encrypted data, such as Electronic Code-Book (ECB) or Counter Mode. When operating in feedback modes such as Ciphertext Feedback mode, the ciphertext of one block must be available before the next block can be encrypted. For the remainder of our discussion, single-block mode will denote feedback modes and multi-block mode will denote non-feedback modes.

3 4 The TMS32 C62x igital Signal Processor

We chose the TMS320C6201 fixed point digital signal processor out of the TMS320C62x family. In this subsection we introduce the key architectural features of the DSP which are relevant for our implementation.

The 'C6201 performs up to 1600 million instructions per second (MIPS) at a clock rate of 200 MHz. These processors have thirty-two 32-bit registers and eight independent functional units. As shown in Figure 1, the 'C62x has four pairs of functional units. The architecture of the DSP has effectively been divided in two identical halves. Each half is composed of four independent functional units (S, , , and) and a bank of sixteen 32-bit registers. The processor also allows limited communication between the two halves.

The multiplier unit is indicated by and accepts two 16-bit words as an input and outputs a 32-bit result. In addition to the two multipliers, the processor provides six arithmetic logic units (ALUs). The unit, that has the ability to perform 32/40-bit arithmetic operations, comparisons, normalization count for 32/40-bits, and 32-bit logical operations. With the unit we can add 32-bit words, subtract, do linear and circular address calculation, and write to and load from memory. The S unit provides the functionality for 32-bit arithmetic operations, 32/40-bit shifts, 32-bit bit-field operations, 32-bit logical operations, branching, constant generation, and register transfers to/from the control register file [16].

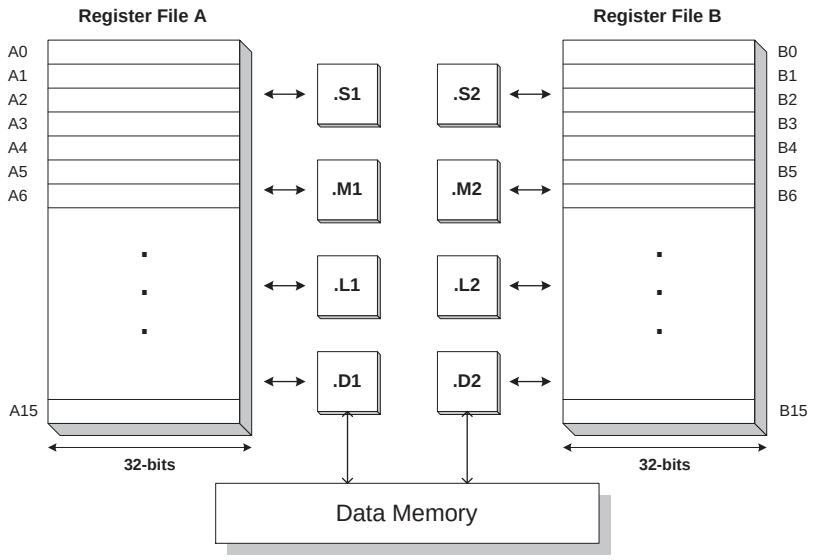


Figure 1: TMS32062x Functional Units [16]

The 'C6201 includes a bank of on-chip memory and a set of peripherals. Program memory consists of a 64K-byte block that is configurable as cache or memory-mapped program space. A 64K-byte block of RAM is used for data memory. The peripheral set includes two serial ports, two timers, a host port interface, and an external memory interface.

The 'C6000 development environment includes: a C Compiler, an Assembly Optimizer to simplify programming and scheduling, and the Code Composer Studio^{TM}, which is a MS Windows debugger interface for visibility into source execution. All of the 'C6000 devices are based on the same CPU core featuring elociTI^{TM}, a highly parallel architecture that provides software-based exibility and good code performance for multi-channel and multi-function applications.

4 Results

4 1 Results on the TMS32 C62 1 SP

All the figures presented in this section refer to a 128-bit block encryption or decryption with a key of 128 bits. The algorithms are timed with the Code Composer Simulator, which is part of the Code Composer Studio^{TM} for the TMS320C6201 DSP. Code Composer Simulator uses the simulated on-chip analysis of a DSP to gather profiling data.

The reported results in Table 1 refer to either a C or a Linear Assembly implementation. In the cases where we had the possibility to choose between two implementations we referenced the fastest results found by us. All the timings shown are obtained from a C implementation using the compiler version 4.0 alpha unless otherwise indicated.

To convert cycle counts into encryption or decryption rates expressed in bits per second, we divided $128 * 200 * 10^6$ by the cycle count. For example, the encryption speed of Twofish in multi-block mode is computed as: $128 * 200 * 10^6/184 = 139.1$ Mbit/sec.

The order of the algorithms is based on the mean speed of encryption and decryption in multi-block mode. The mean speed can simply be calculated by adding the speed of the encryption and decryption functions and then dividing the sum by two. For instance, the mean speed in multi-block mode for RC6 equals $(128.0 + 116.4)/2 = 122.2$ Mbit/sec.

		DSP multi-block mode 200MHz		DSP single-block mode 200MHz		Pentium-Pro 200MHz	DSP multi-block mode/Pentium
		cycles	Mbit/sec	cycles	Mbit/sec	Mbit/sec	
Twofish	encryption	184	139.1	308	83.1	95.0 [17]	1.5
	decryption	172	148.8	290	88.3	95.0 [17]	1.6
RC6	encryption	200 †	128.0	292	87.7	97.8 [12]	1.3
	decryption	220 †	116.4	281	91.1	112.8 [8]	1.03
Rijndael	encryption	228 ‡	112.3	228 ‡	112.3	70.5 [8]	1.6
	decryption	269 ‡	95.2	269 ‡	95.2	70.5 [8]	1.4
Mars	encryption	285	89.8	406	63.1	69.4 [8]	1.3
	decryption	280	91.4	400	64.0	68.1 [8]	1.3
Serpent	encryption	772	33.2	871 *	29.4	26.8 [8]	1.2
	decryption	917*	27.9	917 *	27.9	28.2 [8]	1.0

Table 1: Performance results of the AES candidates on the TMS320C6201

Here are comments about the results in Table 1:

- The highest level of optimizations were used for all algorithms, with the exception of Serpent decryption. The loop in Serpent is too complex and too long so the optimizer was only able to schedule the code in a lower level. Hence, the performance figures for decryption are slightly worse than the numbers for encryption. In addition, the throughput of the decryption function is the same for single-block and multi-block modes.

- The linear assembly code of Rijndael can be optimized by the tools very efficiently. In this case we could not gain a performance advantage by parallel processing, which results in the same speed for single-block and multi-block modes.

- In all cases, except for RC6 encryption, we encrypted and decrypted two blocks at a time in multi-block mode. We were able to process three blocks at a time in parallel for RC6

*C implementation using compiler version 3.0

†Linear assembly implementation using compiler version 3.0

‡Linear assembly implementation using compiler version 4.0 alpha

encryption. Hence, we could use a large number of functional units in parallel and could reach a high throughput. For some of the other algorithms we tried to use three blocks in parallel as well. However, the optimizer was not able to create efficient loops due to the number of instructions.

4.1.1 Results in Multi-Block Mode

In Table 1 we compare the throughput speeds of the TMS320C6201 and a 200MHz Pentium Pro. In order to allow for an easy comparison we added the rightmost column to the table, where we divided the highest speed in multi-block mode on the DSP with the performance numbers on the Pentium. In this way we normalized our numbers by the speed achieved on the Pentium Pro platform. If the ratio is larger than one, the implementation of the algorithm on the DSP is faster than the one on the Pentium. One can see that in all cases but one we could achieve higher throughput on the DSP than the best known results on a Pentium Pro II with the same clock rate. Only for Serpent decryption were the Pentium and the DSP speeds almost identical.

We can also see from the performance ratio in the rightmost column how well the algorithm structure is suited for the DSP. Rijndael encryption and Twofish decryption gain the most when implemented on the DSP compared to the implementation on a Pentium. In both cases the quotient of the throughputs is approximately 1.5, which means that the speed of the particular function on the DSP is roughly 50 faster than the same function on the Pentium.

In addition to our above analysis, we ranked the AES finalists based on their performance on the 'C6000 DSP family. This ranking compares the mean speed of the algorithms in multi-block mode. Twofish with a mean speed of 144.0 Mbit/sec and RC6 with 122.2 Mbit/sec are the fastest algorithms. These two algorithms are followed by Rijndael with a mean throughput of 103.8 Mbit/sec and Mars with 90.3 Mbit/sec. Serpent with 30.6 Mbit/sec is poor in terms of throughput on the DSP.

4.1.2 Results in Single-Block Mode

The results stated above refer only to the cases in which we used multi-block mode. If we look at the single-block mode case, Rijndael encryption and decryption as well as Serpent encryption perform better on the DSP than on a Pentium. Rijndael encryption with 112.3 Mbit/sec is almost 60 faster than the corresponding Pentium implementation and Rijndael decryption at 95.2 Mbit/sec is almost 40 faster. Judged by their speed performance on the C62x, Serpent decryption, Mars encryption and decryption, and Twofish decryption are slightly worse than on a general-purpose computer. The remaining functions, Twofish encryption and RC6 encryption and decryption, are much slower than the corresponding Pentium functions.
If we had ranked the algorithms based on their mean speed in single-block mode, Rijndael with 103.8 Mbit/sec would be the fastest, followed by RC6 with 89.4 Mbit/sec, and Twofish with 85.7 Mbit/sec. Mars with 63.6 Mbit/sec and Serpent with 28.7 Mbit/sec are not as good in single-block mode.

We would like to point out that all of our "best" results were achieved using the methodology described above, and that other coding styles, such as pure assembly, might be able to achieve higher throughputs.

4.1.3 Comparison of the Results with the Critical Path of the Algorithms

Craig S.K. Clapp analyzes the critical path of Crypton, E2, and the five AES finalists. In his analysis, [5] only counts instructions and cycles associated with the transformation of a plaintext block into a ciphertext block in ECB mode. In other words, instructions associated with loading of plaintext, storing of ciphertext, and loop overhead are ignored. Clapp concludes that based on the length of its critical path, Rijndael stands well ahead of the pack with 71 cycles/block. Twofish (162 cycles/block), RC6 (encryption with 181 cycles/block and decryption with 161 cycles/block), and Mars (214 cycles/block) form the second tier. Finally, Serpent's critical path is a factor of two longer than the next nearest candidate (encryption with $\leq$ 526 cycles/block and decryption with $\leq$ 436 cycles/block).

The results that we achieved in single-block mode are in agreement with those obtained by analyzing the critical path. Rijndael is in both cases by far the fastest algorithm. The throughput of RC6 is slightly better than the throughput of Twofish on the DSP, even though the critical path of Twofish is a little shorter than the one from RC6. Mars is ranked in both, the DSP speed analysis and the critical path analysis of [5], the same. Serpent results trail the nearest candidate in both analyses by more than a factor of two. It is important to point out that while the critical path for decryption is shorter than that for encryption in Serpent, decryption is actually slower than encryption in the DSP implementation.

The discrepancies are due to our use of automatic optimization. The optimizer tries to create the best machine code possible. Nevertheless, the optimizer might not be able to reach the cycle count of the critical path for some algorithms. We might be able to overcome these differences by rewriting the functions in full assembly. We were not able to do this because of time constraints.

4.1.4 Memory Usage

Embedded system applications have often memory constrains. Hence this subsection looks at the memory requirements of our implementation. The 'C6201 has three 16 Mbyte regions of external memory. These regions can support synchronous or asynchronous 32-bit access. There is also one 4 Mbyte region of asynchronous external memory which is typically used to store the boot information. The 'C6201 contains one megabit of internal RAM which is split between program and data memory. All this internal memory is zero wait-state. Table 2 summarizes the memory usage of the algorithms in our implementation.

As it can be seen from Table 2, the memory usage of the algorithms varies almost by an order of magnitude. RC6 uses the least program memory and Serpent the most. In some cases, e.g. for Serpent, the algorithms require a large amount of program memory, because we optimized them for speed. Hence we calculated the look-up tables on the " y" with boolean-algebra and this increases

	Memory Usage multi-block mode		Memory Usage single-block mode	
	Data ROM /Bytes	Program /Bytes	Data ROM /Bytes	Program /Bytes
Mars	3072		3072	
encryption		3280		2428
decryption		2956		2372
RC6	0		0	
encryption		608		576
decryption		672		576
Rijndael	16384		16384	
encryption		2360		1180
decryption		2960		1480
Serpent	0		0	
encryption		5844		3568
decryption		6016		5104
Twofish	168		168	
encryption		1416		700
decryption		1420		708

Table 2: Memory Usage on the TMS320C6201

the program code. The data ROM represents constant arrays, which in our cases correspond to the look-up tables. RC6, for example, uses no tables, hence the data ROM is zero.

4 2 Results on the TMS32 C64x

The TMS320C64x clock can be scaled to up to 1.1 GHz and can perform up to 8800 MIPS. The C64x has extended parallelism support with quad 8-bit and dual 16-bit operations. Also, the sixty-four 32-bit registers and 8 functional units lead to better performance. We also took advantage in our implementation of the better data access and the extended instruction set of the C64x (for example, rotation, Galois field multiplication, etc.).

We chose RC6 to be implemented on the C64x. The results that we present in this section are based on a C implementation and are compiled with compiler version 4.0 beta.

The results in Table 3 for RC6 achieved with the 'C64x in multi- and single-block mode are better than the results we got from the 'C6201. RC6 encryption in multi-block mode is almost 70 faster than on a general-purpose machine.

At this point it is important to remark that the optimizer tools are quite advanced for the 'C62x, but are still in a very early stage for the 'C64x. That means if we only perform C code optimizations,

we will not get good performance numbers on the 'C64x. We expect an improvement when we rewrite the functions in linear assembly. We did a detailed analysis for hand coded assembly RC6 and we estimated a performance of 229 cycles/block (for each encryption- and decryption-function) in single-block mode.

		DSP multi-block mode 200MHz		DSP single-block mode 200MHz		Pentium-Pro 200MHz	DSP multi-block mode/Pentium
		cycles	Mbit/sec	cycles	Mbit/sec	Mbit/sec	
RC6	encryption	155	165.2	277	92.4	97.8 [12]	1.7
	decryption	154	166.2	278	92.1	112.8 [8]	1.5

Table 3: Performance results of two AES candidates on the TMS320C64x

5 Conclusions

"How well are high-end DSPs suited for the AES algorithms?" was the main question that we asked ourselves as a motivation to write this paper. We noticed that in almost all cases the AES finalists' encryption and decryption functions reach higher speeds on the 'C6000 DSPs than the best known Pentium Pro II implementations, at identical clock rates. It was observed that some of our implementations on the 'C6201 were over 50 faster than the best known performance numbers on the Pentium platform. In addition, our implementation of RC6 on the 'C64x reached speeds which were almost 70 faster than those of the Pentium. RC6 on the 'C64x encrypts with a throughput of 165.2 Mbit/sec and decrypts with a speed of 166.2 Mbit/sec. Twofish with an encryption speed of 139.1 Mbit/sec and decryption of 148.8 Mbit/sec was by far the fastest throughput that we obtained on the 'C6201. Hence, we can conclude from our results, that state-of-the-art DSPs are well suited for the architecture of the AES finalists.

6 Acknowledgment

We would like to thank William Cammack from TI for his helpful comments.

References

[1] Second Advanced Encryption Standard (AES) Conference. Rome, Italy, March 1999. National Institute of Standards and Technology (NIST).

[2] R. Anderson, E. Biham, and L. Knudsen. Serpent: A Proposal for the Advanced Encryption Standard. In *irst Ad anced ncryption Standard A S Conference*, entura, CA, 1998.

[3] P. Barrett. Implementing the Rivest Shamir and Adleman Public Key Encryption Algorithm on a Standard Digital Processor. In A. M. Odlyzko, editor, *Ad ances in Cryptology Crypto* , volume 263, pages 311 326, Berlin, Germany, August 1986. Springer- erlag.

[4] Carolynn Burwick, Don Coppersmith, Edward D'Avignon, Rosario Gennaro, Shai Halevi, Charanjit Jutla, Stephen M. Matyas Jr., Luke O'Connor, Mohammad Peyravian, David Safford, and Nevenko unic. Mars - a candidate cipher for AES. In *irst Ad anced ncryption Standard A S Conference*, entura, CA, 1998.

[5] Craig S.K. Clapp. Instruction-level Parallelism in AES Candidates. Second AES Conference, March 1999. `http://csrc.nist.gov/encryption/aes/reound1/conf2/papers/clapp.pdf`

[6] J. Daemen and . Rijmen. AES Proposal: Rijndael. In *irst Ad anced ncryption Standard A S Conference*, entura, CA, 1998.

[7] Stephen R. Dusse and Burton S. Kaliski Jr. A Cryptographic Library for the Motorola DSP56000. In Ivan B. Damgard, editor, *uroCrypt* , volume 473 of *ecture otes in Computer Science*, pages 230 244, Berlin, Germany, May 1990. Springer- erlag.

[8] Brian Gladman. AES Algorithm Efficiency, 2000. `http://www.btinternet.com/~brian.gladman/cryptography_technology/Aes2/index.htm`

[9] Karol Gorski and Michal Skalski. Comments on the AES Candidates. Technical report, National Institute of Standards and Technology, ENIGMA SOI Sp. z o.o., Warsaw, Poland, April 1999. `http://csrc.nist.gov/encryption/aes/round1/comments/R1comments.pdf`

[10] Kouichi Itoh, Masahiko Takenaka, Naoya Torii, Syouji Temma, and asashi Kurihara. Fast Implementation of Public-Key Cryptography on a DSP TMS320C6201. In Cetin K. Koc and Christof Paar, editors, *Cryptographic ardware and mbedded Systems*, volume 1717 of *ecture otes in Computer Science*, pages 61 72, Berlin, Germany, August 1999. Springer- erlag.

[11] R. Rivest, M.J.B. Robshaw, R. Sidney, and .L. in. The RC6™ Block Cipher. In *irst Ad anced ncryption Standard A S Conference*, entura, CA, 1998.

[12] RSA Security. The RC6 Block Cipher - Performance, 1999. `http://www.rsasecurity.com/rsalabs/aes/rc6_performance.html`

[13] B. Schneier, J. Kelsey, D. Whiting, D. Wagner, and C. Hall. Twofish: A 128-Bit Block Cipher. In *irst Ad anced ncryption Standard A S Conference*, entura, CA, 1998.

[14] W. Stallings. *Cryptography and etwork Security.* Prentice Hall, Upper Saddle River, New Jersey 07458, 2nd edition, 1999.

[15] Texas Instruments Incorporated. *S C ptimi ing C Compiler ser s uide*. Custom Printing Company, Owensville, Missouri, February 1998.

[16] Texas Instruments Incorporated. *S C C rogrammer s uide*. Custom Printing Company, Owensville, Missouri, February 1998.

[17] D. Whiting. Twofish Timing Measurements. electronic mail personal correspondence, January 2000.

Fast Implementations of AES Candidates

Kazumaro Aoki[1] and Helger Lipmaa[2]

[1] NTT Laboratories
1-1 Hikarinooka, Yokosuka-shi, Kanagawa-ken, 239-0847 Japan
maro@isl.ntt.co.jp
[2] Küberneetika AS
Akadeemia tee 21, 12618 Tallinn, Estonia
helger@cyber.ee

Abstract. Of the five AES finalists four—MARS, RC6, Rijndael, Twofish—have not only (expected) good security but also exceptional performance on the PC platforms, especially on those featuring the Pentium Pro, the NIST AES analysis platform. In the current paper we present new performance numbers of the mentioned four ciphers resulting from our carefully optimized assembly-language implementations on the Pentium II, the successor of the Pentium Pro. All our implementations follow well-defined API and timing conventions and sensible guidelines, like no using of self-modifying code and key-specific static data — i.e., tricks that speed up the implementation but at the same time restrict the field of application. Our implementations are up to 26% percent faster than previous implementations. Our work also shows how a simple change (inclusion of the MMX technology) in the analysis platform can influence the relative encryption speed of different ciphers. To enable everyone to compare their implementations to ours, we also fully specify our procedures used to obtain the speed numbers.

1 Introduction

For more than 20 years, DES [FIP77] has been a widely employed cryptographic standard. While the best cryptanalytic attacks against DES (differential and linear cryptanalysis) are still highly impractical, during the last years DES has became obsolete for its too short key and block sizes, not withstanding the current advances in computing technology. Motivated by this, NIST initiated a new effort to replace DES as a standard. 21 algorithms were submitted and 15 algorithms were accepted as AES (*Advanced Encryption Standard*) candidates, of which 5 candidates—MARS [BCD+98], RC6 [RRSY98], Rijndael [DR98], Serpent [ABK98], Twofish [SKW+99b]—were chosen to the second round.

However, the AES process was started not only due to the theoretical reasons: there are a few well-known constructions, including 3DES, that seem to have very good security margins. Unfortunately, 3DES, based on the hardware-oriented DES, is unsatisfyingly slow on the modern 32- and 64-bit computer architectures: modern block ciphers are up to 10 times faster than 3DES. Regardless of these ciphers having unproven (even by time) security properties, they are widely used in the industry by pragmatic reasons: hardware applications like 1 GBits/s Ethernet or on-the-fly encryption of 160 MByte/s

SCSI hard disks are requesting for faster ciphers. Clearly, the situation of having a (moderately) secure and (moderately) fast *de jure* standard DES, a (probably) secure and (clearly) slow *de facto* standard 3DES and some fast but with unknown security margin *de facto* standards is not acceptable: there should be a single standard that is both secure and fast. This is one of the reasons why, when inviting the public to propose candidates for the AES, NIST explicitly stated that the new standard should be both "more secure and faster" than 3DES.

While security of the candidates cannot be exactly quantified by the currently known methods, it seems to be easier to measure their speed. However, there is still a lot of ambiguity in answering the question what AES candidate is the fastest. Several papers (including [Lip99,SKW+99a]) have compared AES candidates speed, but since the implementations quoted in them are often incomparable (or based on pure estimations), one cannot make direct conclusions about the efficiency of the ciphers based on the published papers. Incomparability stems from the different implementation assumptions, API's, hardware (e.g., processors) and software (e.g., compilers) used by implementers. Even more, some of the timings presented in previous papers correspond to "show-case" (as opposed to practically applicable) implementations, some examples of those being the fastest implementation of Twofish [SKW+99b] that uses self-modifying code and Brian Gladman's implementations of AES candidates [Gla99] that use a number of key-specific static variables instead of allocating a register to address them, therefore effectively freeing some registers for other uses. Especially in the case of the Pentium family, where the number of available registers is very restricted, such implementations may result in a huge speed up. However, both types of implementation tricks restrict the application area of the implementation.

In the current paper we try to give a satisfactory answer to the question "what cipher is the fastest on the Pentium II" by carefully optimizing the 4 fastest AES candidates—MARS, RC6, Rijndael and Twofish—in Pentium II assembly, using for all implementations exactly the same, reasonable in practice, API and speed measurement conditions for all the ciphers. Due to this, our results are much fairer than most of the previously known ones: our implementations can be seen as black boxes applicable in almost any possible application of block ciphers on an environment featuring Pentium II. Additionally, careful optimization process resulted in implementations that are clearly faster than the previously known implementations. (Except for Twofish, which has still a faster "show-case" implementation.)

We start the paper by describing our platform of choice (Section 2), implementation philosophy and API (Section 3). Section 4 briefly surveys our results, and Section 5 gives more details on the problems encountered when implementing the ciphers. More information about the Pentium II is given in the Appendices.

2 Choice of the Platform

Our first principal choice was the decision what processor to use. By purely pragmatic reasons we decided that the implementation environment equips an Intel Pentium family CPU: while this family is not the most modern processor family available, it is the most widespread one at the moment of writing this paper and most probably also during the

next few years. Therefore, since in the foreseeable future most of the software-based commercial security applications run on the Pentium family (as recognized also by the AES finalists designers), this family has the most direct impact on the choice of a cipher by security consumers.

At second, from the Pentium family we decided to choose the Pentium II processor. At first, it is a more advanced processor than Pentium Pro, the NIST AES analysis platform: the Pentium II provides (twice) larger register space due to the added MMX technology, and many new MMX-specific commands. Compared to the Pentium Pro, the Pentium II is also easier to obtain at the current stage, since Pentium Pro has been out of the manufacturing for a while. On the other hand, the Pentium II was preferred by the authors to the Pentium III since the latter is somewhat too new and controversial due to the privacy issues.

Another reason to choose Pentium II was that as the successor of the NIST AES analysis platform, implementing the AES candidates on the Pentium II could provide some insights on how generally suitable are the candidates, some of which were specifically optimized for the Pentium Pro, on future processors having features unpredicted by algorithm designers. While this is not as crucial as withstanding the "future attacks", it still gives some ideas about the possible longevity of the cipher. (We clearly would not want the AES in 20 years to have the role the 3DES has today!)

As shown in [Lip98], the MMX technology can seriously speed up IDEA ([LM90], [LMM94]), one of the believably most secure block ciphers with 64-bit block size. As stated in [Lip98], this can be done since IDEA has its key attributes similar to those of multimedia applications, for which the MMX technology was originally created. An open question posed in [Lip98] was how much would the MMX technology help implementing other ciphers, including the AES candidates. In the following we will partially answer to that question, showing that also some ciphers using only "simple" operations can greatly benefit from the added MMX technology. A short overview of Pentium II that is necessary for implementers and for cryptographers who design ciphers optimized for this platform is given in Appendix A. We refer for Intel manuals for a more complete overview.

3 Implementation Considerations

Several papers (including, in particular, [Lip99,SKW+99a]) have compared AES candidates speed, but since the implementations quoted in them are often incomparable (or based on pure estimations), one cannot make direct conclusions about the efficiency of these algorithms based on the published papers. Incomparability stems from the different implementation assumptions, API's, hardware (processors) and software (compilers) platforms used by implementers. Even more, some of the numbers there correspond to the "show-case" (as opposed to practically applicable) implementations; including the bizarre case that one candidate was claimed to be the fastest on its inventors laptop under some suitable conditions.

As another example of the unsuitability of some "show-case" implementations, the fastest implementation of Twofish [SKW+99b] uses self-modifying code and therefore cannot be used in a number of applications, while Brian Gladman's implementations of

AES candidates [Gla99] use a number of key-specific static variables instead of allocating a register to address them, therefore effectively freeing some registers for other uses. Especially in the case of the Pentium family, where the number of available registers is very restricted, such implementations may result in a huge speed up. On the other hand, Gladman's implementations cannot be used several applications, including multithreaded programs and SMP (symmetric multi-processing) systems.

Most of the security customers need however speed numbers applicable in whatever product they use in whatever environment in runs (for example, in a Linux kernel-supported IPSEC implementation, secure login or multithreaded access to encrypted storage arrays). For users it is necessary to know in what environment the measured speed numbers were obtained, to be able to calculate the possible efficiency of the ciphers in their own environments. Additionally, full specification is important for other implementers to be able to compare their implementations with ours. Hence, apart from providing "clean" implementations under some reasonable public assumptions, we shall also next fully specify these assumptions:

- We do not use self-modifying code ("code compilation" [SKW+99b]) since it makes the implementation inapplicable in a number of situations, e.g., in operation-system kernel and ROM-based applications.
- We additionally decided not to use key-specific static areas since then the implementation could not be used, e.g., in SMP-capable systems and multithreaded programs.
- We decided to maximally use the MMX technology since it should not be forbidden in any reasonable modern environment. (While using self-modifying code and key-specific static areas is generally considered to be a bad programming practice.)
- We decided to use exactly the same API (specified later in Section 3.1) in all our implementations.
- A number of well-understood assumptions that 1) improve the speed and can be easily followed by implementers or 2) are essential to even be able to measure the speed:
  - All codes and data are correctly aligned.
  - Input and output texts and codes are preloaded to L1 cache in the possible extent to reduce the number of cache misses.
  - Simplicity of code: we tried to reduce time spent during writing and optimizing the code. In particular, all our implementations use highly optimized but round-number independent round macros. (Hence, our results could be slightly bettered if every round would optimized separately to avoid, e.g., delays in fetching stage.)

3.1 API

Since a different API can be influence the speed of an implementation severely, we also decided to fully specify the API used by us to make for the other implementers easier to compare their implementations to the ours. We felt that this is necessary, since AES candidate implementations reported in [Lip99] vary greatly in their API's.

```
void xxKS(char *master, uint32 bitLen, char *eKey);
void xxEnc(char *inBlk, uint32 lenBlk, char *eKey,
           char *outBlk);
void xxDec(char *inBlk, uint32 lenBlk, char *eKey,
           char *outBlk);
```

where

xx is algorithm name (e.g., `Rijndael`).
xxKS is key scheduling subroutine.
xxEnc is encryption subroutine that encrypts `lenBlk` blocks of plaintext starting from the address `inBlk` to the ciphertext location `outBlk`, by using extended key `eKey`, in ECB block cipher mode.
xxDec is decryption subroutine with the same input conventions as `xxEnc`.
uint32 is the type of 32-bit unsigned integers (in the case of Pentium II, equal to `unsigned long` in the case of most compilers).
master is pointer to the master key bits.
bitLen is the bit length of a master key.
eKey is pointer to subkeys and other initialization data, used later by encryption and decryption.
inBlk is pointer to input texts to be encrypted in the case of `xxEnc` and to be decrypted in the case of `xxDec`.
outBlk is pointer to the corresponding output texts.
lenBlk is number of blocks to be encrypted or decrypted.

Fig. 1. Specification of our API.

Note that our API, depicted in Figure 1, is essentially equivalent to the API's used in most of the commercial applications, specifying only those inputs and outputs to the algorithms that are really needed by the algorithms. (Names of the subroutines and their parameters of course do not affect the speed, of course.) Our API was fixed for the key length of 128-bits due to the feeling that at the time when greater key sizes become necessary, our implementation platform would already be a history.

Here, the key schedule and decryption subroutines are specified only for completeness. Since in the current paper we are not interested in the optimization of these subroutines, we almost do not mention decryption and key schedules hereafter.

3.2 How to Measure a Number of Cycles

Different time measurement methods may change the speed numbers quite dramatically. As in the case of the API's, we decided to use one, sensible published and *fully specified* convention (specified in Figure 2) for all the implementations. (Note that this wrapping corresponds almost exactly to the method specified in [Fog00], to which the reader is referred for a throughout explanation of the method.) The inputs and key of the cipher are generated randomly before the measurement begins, to prevent "optimization" for specific class of keys. The input variable `lenBlk` was chosen to be equal to 8000 so that the input and output texts would not fit in the L1 cache. Also, `time` is a work area of type `uint32`, used in later calculations.

```
movd mm0, dword ptr [time];  /* warm cache and set MMX state */
xor eax, eax;
cpuid;                       /* serialize instructions */
rdtsc;                       /* read time-stamp counter */
mov dword ptr [time], eax;   /* save counter */
xor eax, eax;
cpuid;                       /* serialize instructions */
/* xxEnc() or xxDec() */
xor eax, eax;
cpuid;                       /* serialize instructions */
rdtsc;                       /* read time-stamp counter */
sub dword ptr [time], eax;   /* compute the difference */
emms;                        /* empty MMX state */
```

Note that `time` is a 4 bytes work area.

Fig. 2. Time measurement code

```
   /* push all used registers */
   cmp dword ptr [lenBlk], 0;
   jz L1;
   align 16;
L0:
   dec dword ptr [lenBlk];
   jnz L0;
L1:
   /* pop these registers once more */
```

Fig. 3. Null function

Note that this method has some overhead, due to both high latency of the `rdtsc` instructions and also the overhead caused by looping instructions like `jnz` which are not formally part of the cipher itself. (Looping instructions can be seen as a part of the block cipher mode, however.) We measure this overhead by using the null function shown in Fig. 3 obtaining a value `nulltime`, and then we subtract it from the value of `time` obtained by measuring the speeds of different encryption/decryption procedures. Finally, this result is divided by the number of blocks encrypted. Intuitively, by using this method we obtain the number of cycles corresponding to unrolled implementation of the block cipher, or to the implementation where we only care about the time encrypting one block takes without adding any extra overhead. (Note that the subtracted overhead number was equal to ≈ 6 in the case $n = 8000$. One could easily add this number to those presented later to get the number of cycles *with* overhead.)

Chosen time measurement method is also reasonable in practice: when the value of `lenBlk` was chosen to be different, for most of the implementations (*including* the implementation of null cipher), the execution times increased by almost the same constant. Hence, the null cipher proved experimentally to be well-defined.

Cipher	Mbits/s on a 450 MHz Pentium II	Cycles per block	Best previous result	Speedup
Null cipher	—	6	—	—
RC6	258 Mbits/s	223	243 [Riv98]	8%
Rijndael	243 Mbits/s	237	320 [DR98]	26%
Twofish	204 Mbits/s	282	315 [SKW$^+$99b]	11%
MARS	188 Mbits/s	306	390 [BCD$^+$98]	22%

Table 1. Performance in clock cycles per block of output of four AES finalists. (Only encryption considered)

Finally, we did a loop of 500 times over the described measurements and then chose the smallest number for every cipher, since that corresponds most likely to the case where most of the data and code are in L1 cache and the branch prediction works successfully: i.e., to the bulk encryption speed of the cipher itself.

4 Implementation Results

From the five AES finalists, one (Serpent) is regarded as a very conservative design but at the same time also being clearly slower than the other AES finalists. Rest of the finalists have comparable timings on most of the modern computer platforms, where one of the ciphers is the fastest in one platform, and another one in another platform. Since also on the Pentium II processor, Serpent seems to be very slow by the published data, we decided postpone its implementation to the future and concentrate on the fast ciphers.

Timings, obtained by measuring the speed of implementations by following previously specified procedures are summarized in Table 1[1]. The numbers in the middle columns show how many cycles it takes to encrypt one 128-bit block by using the chosen cipher with a 128-bit key. These results indicate that the chosen four AES finalists are extremely fast. For comparison, the standard hash algorithm SHA-1 *hashes* a 512-bit block in 837 cycles (i.e., 13.1 cycles per byte) and DES and 3DES encrypt a 64-bit block respectively in 340 and 928 cycles (resp., 42.5 and 116 cycles per byte) [PRB98], while RC6 and Rijndael respectively encrypt a 128-bit block in 223 and 237 cycles (resp., 13.9 and 14.8 cycles per byte). However, note that the cited timings in [PRB98] were obtained on a plain Pentium and therefore could most probably be improved on the Pentium II.

Our results seem to indicate, that the speed difference between different ciphers is less than expected: as before, RC6 is still the fastest cipher on the Pentium II, but the difference between it and Rijndael has decreased seriously. Hence we hesitate to say that RC6 is the fastest cipher. However, based on the cited results, we can classify the ciphers to two groups: blastingly fast ciphers RC6 and Rijndael and somewhat slower, but still very fast ciphers Twofish and MARS.

[1] We also started to code the decryption routines, finishing RC6 decryption (209 cycles per block) and Twofish decryption (276 cycles per block).

However, one has to keep in mind that RC6 and MARS have design features that make them specifically efficient on the Pentium Pro (and its successors), while their performance seriously degrades on other processors [Lip99,SKW+99a]. This is due to the use of complex instructions (32-bit multiplication and data-dependent rotation) that are cheap on the P6 family (Pentium Pro, Pentium II, Celeron, Xeon and Pentium III) but very expensive on most of the other platforms. Interestingly, also the next generation Pentium processor (code-named "Willamette", [Int00]) has latency 10 multiplication and latency 2 or 4 shifts, as compared to latency 4 multiplication and latency 1 shifts on the P6 family [Int00, Section 4.1.3]. Hence, RC6 and MARS would considerably slow down on the Willamette, the next generation Pentium family processor. On the other hand, Rijndael and Twofish are based on simple operations, and run equally well on all platforms. The speed ratio between Rijndael and Twofish seems be remain *almost* the same on the other platforms [Lip99] (namely, Rijndael being $5 \ldots 25\%$ faster than Twofish).

Note that the speed up percents in Table 1 correspond to the achieved speed ups compared to the fastest "clean" implementations (i.e., those not using key-specific static data or self-modifying code). However, these percents do not always mean that our implementation techniques were exactly as much better. For example, the best previous implementation of Rijndael was done for the plain Pentium, but not for the Pentium Pro: a factor that may have negatively affected its performance. The best previous "clean" implementation of MARS was written in C, and therefore had also a relatively slow performance. However, our own C implementation of MARS is clearly faster than the one given in Table 1. In the case of Rijndael, most of the acceleration Rijndael is due to the efficient use MMX technology. In general, speed up comes mainly from better optimization (elaborated tradeoff between processor operating stages) and full usage of the Pentium II possibilities (i.e., the MMX technology).

To further clarify how does the Pentium II architecture impact the speed, Table 2 shows the detailed information of our implementations in encryption mode in the micro-operation level. Usage of the table is simple. For example, in the intersection point of "@round" row and "port 01" column in `TwofishEnc` table one would find 19. That means that there are 19 μoperations in the round function of `TwofishEnc` which will be executed on port 0 or port 1.

Interestingly, our implementations of MARS, Rijndael and Twofish all require approximately the same number of μoperations, while RC6 is about two times "better" in this category. On the other hand, RC6 is also the worst cipher to parallelize: while in Rijndael, more than 2.5 μoperations are executed per a cycle, RC6 can only mildly use the super-scalar parallelism of Pentium II. More cipher-specific comments will be given in the next.

5 Cipher-Specific Comments

5.1 MARS

In the case of MARS [BCD+98], the speed difference between a carefully optimized C implementation (using a recent snapshot of the `gcc` compiler) and an optimized assembly language implementation is only about 11% on the Pentium II. The speedup

	port 0	port 1	port 01	port 2	port 3	port 4	total
MARS encryption							(1.87 μops/cycle)
prewhitening			5	8			13
forward mixing	16		77	32			125
@core ($\times 16$)	6		9	3			18
backward mixing	16		85	32			125
postwhitening		1	8	4	4	4	21
total	128	1	319	124	4	4	572
RC6 encryption							(1.47 μops/cycle)
prewhitening			2	7			9
@round ($\times 20$)	8		5	2			15
postwhitening		1	4	5	5	5	20
total	160	1	106	52	5	5	329
Rijndael encryption							(2.54 μops/cycle)
whitening		1	8	6			15
@round ($\times 9$)	4	1	34	19			58
last round	4	3	31	20	3	3	64
total	40	13	345	197	3	3	601
Twofish encryption							(2.11 μops/cycle)
prewhitening			5	8			13
first round	5		19	10			34
@round ($\times 15$)	6		19	10			35
postwhitening	2	1	8	4	4	4	23
total	97	1	317	172	4	4	595

Table 2. Number of μoperations in our implementations

comes mainly from a slightly more efficient allocation of the integer registers and some (minimal) usage of the MMX instructions in the assembly implementation. However, the MMX technology is only moderately useful for MARS, since the complex instructions performed in MARS (i.e., 32-bit multiplication, data-dependent rotation and S-box lookups) are not available for the MMX registers. Additionally, due to the high data-dependency there is very limited freedom in meaningfully rescheduling the instructions in MARS, which also means that one cannot avoid all the delays on all the processor operating stages.

Another drawback is that during MARS encryption, some execution ports are considerably more overloaded than others. Namely, more than 78% of μoperations go either to port 0 or 1. The most overloaded is port 0, since 128 μoperations go only to this port — including 16 multiplications and extensively used rotations.

5.2 RC6

From implementers point of view, problems arising when optimizing an RC6 implementation are similar to those arising when coding MARS in many aspects: both ciphers rely on the same complex instructions, have long critical paths and overloaded

port 0. However, since RC6 uses multiplications even more extensively, it is even less parallelizable. Table 2 shows that our implementation includes 160 port 0 μoperations, which includes 40 multiplications with latency 4.

RC6 is a very Pentium II-friendly cipher, and it is very easy to code it even in the assembly language. It can also be very efficiently implemented in C: the speed difference between a C implementation and an assembly implementation is about 18%. (The difference is bigger than in the case of MARS since `gcc`, the test compiler, performs very poorly in translating the quadratic formulas of type $x \cdot (2x + 1)$ to the Pentium II assembly language.) It is straightforward to obtain an optimized assembly language implementation from the C implementation: one does not have many possibilities to reschedule the code.

5.3 Rijndael

As opposed to MARS and RC6, Rijndael [DR98] is not C-friendly (at least not `gcc`-friendly) in the sense that assembly implementation is about 44% slower than `gcc`-implementation of the same cipher. It is however mainly due to the inefficiency of the `gcc` compiler: our implementation of Rijndael makes very heavy use of the MMX technology, but also of 8-bit instructions provided by Pentium family. However, `gcc` cannot efficiently use either of these.

Rijndael can effectively use the MMX since Rijndael is based only on most simple imaginable operations (`load`, `xor`), all of which are supported by the MMX technology. Additionally, since Rijndael has large internal parallelism (at least four-times, but partially up to 16-times parallelism!), there is a large number of possibilities to reschedule its code. Our implementation was obtained by doing so in a way that all the delays in the different stages of the Pentium II operation would be minimized. The final result is very impressive for the Pentium II: it executes 2.54 μoperations per a cycle.

Not the last factor that makes Rijndael suitable for the Pentium II is the fact that almost exactly one third of the μoperations in our implementation of Rijndael go to port 2, while the remaining $2/3$ of μoperations go to ports 0 and 1. Due to this and parallelism we get that during the Rijndael encryption 3 μoperations could be executed in parallel almost all the time. However, this (not to mention other aspects like decoding and fetching delays) also makes 20 cycles per round a lower bound for Rijndael and shows that our result may be very close to the optimal one. To facilitate more efficient implementations, the Pentium II should feature three ALUs, two concurrent memory access ports and also more decoders and retirement units: features that are not cipher-specific and would improve the speed of most of the applications.

Finally, we measured the timings of r-round Rijndael for variable r without any additional fine-tuning: those implementations are unoptimized since they use the same round macros as the 10-round Rijndael without any additional effort to optimize them to reduce, say, fetching delays. In particular it turned out that 8-round Rijndael (essentially equivalent to the cipher Square [DKR97] from the implementers point of view) encrypts a block in 193 cycles. 192-bit Rijndael (12 rounds) took 286 cycles, and 256-bit Rijndael (14 rounds)—333 cycles. Note that since 12-round Rijndael is very similar to Crypton [Lim98], 286 cycles is also a (hopefully) close approximation for the speed of latter.

5.4 Twofish

Twofish is designed to be well-suited on multiple platforms, including also the Pentium II. From the implementers point of view it resembles Rijndael in many aspects, by using only simple instructions but also some large-scale components of the latter (e.g., MDS, to provide diffusion). Due to the use of low-level instructions, Twofish is also relatively slow in C compared to the assembly (the difference is about 37%).

Main difference for implementers between Rijndael and Twofish is the inclusion of the Pseudo-Hadamard Transformation that somehow complicates Rijndael's clear structure and makes it less parallelizable: while the number of μoperations in our implementation of Twofish is less than in our implementation of Rijndael, it turned out to be very difficult to use the MMX technology to optimize Twofish. Hence, Twofish is only moderately parallelizable, although the parallelism of our implementation (2.11 μoperations per cycle) is relatively good.

6 Conclusion and Work in Progress

We achieved the fastest implementations of four of the AES finalists on the Pentium II processor, obtaining speedup $8\% \ldots 26\%$ compared to the previously known implementations. Since all implementations were coded by using the same sensible assumptions, they provide a more adequate efficiency comparison of the AES finalists than the previous papers. We demonstrated that MMX can be quite efficiently used to speedup Rijndael, but is only moderately useful for other ciphers. (However, our implementations depend on the availability of MMX technology to a lesser or greater extent and in general do not run on the Pentium Pro.) We provided full specification on our time-measurement conditions to simplify for the future implementers to compare their implementations to ours.

Our implementations are not the final: we continue optimizing them. Up-to-date results will be available at the AES efficiency table [Lip99].

References

[ABK98] Ross Anderson, Eli Biham, and Lars Knudsen. Serpent: A Flexible Block Cipher With Maximum Assurance. In *The First Advanced Encryption Standard Candidate Conference*, Ventura, California, USA, 20–22 August 1998.

[BCD+98] Carolynn Burwick, Don Coppersmith, Edward D'Avignon, Rosario Gennaro, Shai Halevi, Charanjit Jutla, Stephen M. Matyas Jr., Luke O'Connor, Mohammad Peyravian, David Safford, and Nevenko Zunic. MARS — A Candidate Cipher for AES. Original paper and a tweak to it are available from `http://www.research.ibm.com/security/mars.html`, June 1998.

[DKR97] Joan Daemen, Lars Knudsen, and Vincent Rijmen. The Block Cipher Square. In Eli Biham, editor, *Fast Software Encryption '97*, volume 1267 of *Lecture Notes in Computer Science*, pages 149–165, Haifa, Israel, January 1997. Springer-Verlag.

[DR98] Joan Daemen and Vincent Rijmen. The Block Cipher Rijndael. In *Third Smart Card Research and Advanced Applications Conference Proceedings*, 1998. To appear.

[FIP77] FIPS. Data Encryption Standard. Technical report, U.S. Department of Commerce/National Bureau of Standards, National Technical Information Service, Springfield, Virginia, 1977. FIPS 46.

[Fog00] Agner Fog. How to Optimize for the Pentium Microprocessors. Available from `http://www.agner.com/assem/`, 11 March 2000.

[Gla99] Brian Gladman. AES algorithm efficiency. Unpublished. Information available from `http://www.btinternet.com/~brian.gladman/ cryptography_technology/`, January 1999.

[Int99] Intel. *Intel Architecture Optimization. Reference Manual*, 1999. Order Number 245127-001.

[Int00] Intel. *Willamette Processor Software Developer's Guide*, February 2000. Order Number 245355-001.

[Lim98] Chae Hoon Lim. Specification and Analysis of CRYPTON Version 1.0. Unpublished. Available from `http://crypt.future.co.kr/~chlim/pub/cryptonv10.ps`, 22 December 1998.

[Lip98] Helger Lipmaa. IDEA: A cipher for multimedia architectures? In Stafford Tavares and Henk Meijer, editors, *Selected Areas in Cryptography '98*, volume 1556 of *Lecture Notes in Computer Science*, pages 248–263, Kingston, Canada, 17–18 August 1998. Springer-Verlag.

[Lip99] Helger Lipmaa. AES candidates: A survey of implementations. An on-line table. Information available from `http://home.cyber.ee/helger/aes/`, January 1999.

[LM90] Xuejia Lai and James Massey. A proposal for a new block encryption standard. In I. B. Damgård, editor, *Advances in Cryptology — EUROCRYPT '90*, volume 473 of *Lecture Notes in Computer Science*, pages 389–404. Springer-Verlag, 1991, 21–24 May 1990.

[LMM94] Xuejia Lai, James L. Massey, and Sean Murphy. Markov ciphers and differential cryptanalysis. In D. W. Davies, editor, *Advances on Cryptology — EUROCRYPT '91*, volume 547 of *Lecture Notes in Computer Science*, pages 17–38, Brighton, UK, April 1994. Springer-Verlag.

[PRB98] Bart Preneel, Vincent Rijmen, and Antoon Bosselaers. Recent developments in the design of conventional algorithms. In B. Preneel, R. Govaerts, and J. Vandewalle, editors, *Computer Security and Industrial Cryptography, State of the Art and Evolution*, volume 1528 of *Lecture Notes in Computer Science*, pages 90–115. Springer-Verlag, 1998.

[Riv98] Ronald L. Rivest. Futher Notes on RC6. Unpublished. Available from `http://theory.lcs.mit.edu/~rivest/rc6-notes.txt`, 20 June 1998.

[RRSY98] Ronald L. Rivest, Matt J. B. Robshaw, R. Sidney, and Y. L. Yin. The RC6 Block Cipher. Available from `http://theory.lcs.mit.edu/~rivest/rc6.ps`, June 1998.

[SKW+99a] Bruce Schneier, John Kelsey, Doug Whiting, David Wagner, and Chris Hall. Performance comparison of the AES submissions. Unpublished. Information available from `http://www.counterpane.com/`, January 1999.

[SKW+99b] Bruce Schneier, John Kelsey, Doug Whiting, David Wagner, Chris Hall, and Niels Ferguson. *The Twofish Encryption Algorithm: A 128-Bit Block Cipher*. John Wiley & Sons, April 1999. ISBN: 0471353817.

A Pentium II for Cipher Designers and Implementers

A.1 MMX Technology

The Pentium II has 8 integer (including stack pointer) and 8 new MMX registers; the latter were not present in the Pentium Pro. While there is a great number of operations available on the integer registers, MMX registers are much more "RISCy": only a few instructions affect them, including move, Boolean operations, 16-bit arithmetic and shifts. Available set of instructions does not include several operations used in the modern block cipher design, including rotation and 32-bit multiplication. On the other hand, the MMX technology provides 64-bit versions of Boolean operations and data moves (i.e., the simplest possible operations), and also parallel 4-way addition and multiplication of 16-bit data. 16-bit multiplication is currently used in a very few ciphers, but as shown in [Lip98], ciphers that base their security on extensive use of 16-bit multiplication can be speed up considerably if using the MMX technology.

Despite of MMX's attractiveness, at the current state of the affairs many C compilers (for example, `gcc`, the standard compiler for Linux machines) do not yet produce MMX code. Hence, for the Pentium II the assembly implementations are potentially more efficient than C-language implementations. Partially by this reason, many designers and implementers of AES candidates seem not to know about MMX at all.

A.2 Processor stages.

The Pentium II processor (as other processors in the P6 family) operates in several stages. At first the instructions are fetched from the main memory and then broken down (decoded) into μoperations (simple instructions consist of only one μoperation, while complex instruction have more μoperations). Thereafter, the μoperations go via a short queue to the register allocation table that allows register renaming. After that, instructions go to reorder buffer that enables out-of-order execution. There it stays until the operands it needs are available. Ready-for-execution μoperations are sent to the execution units, and thereafter retired [Int99,Fog00]. During the optimization one has to count on all different stages of processor operation to find a good tradeoff between the delays introduced in them. The technicalities presented hereafter could be most interesting for the implementers, but also for the cipher designers who want to create ciphers optimized for the Pentium II. The most important lesson from the next is that fixing any processor stages (e.g., decoding), suitable reordering of the instructions can considerably reduce the delays at this stage. However, the same reordering usually introduces additional delays in some other stages and therefore, code reordering is always a complicated tradeoff. To achieve really fast implementations, a cipher should have great internal parallelism that provides many different instruction reordering possibilities, from what the best could be found after possibly exhaustive search. Of course, one could design a cipher that would have only one possible order of instructions, optimized specifically for Pentium II. However, such cipher could slow down severely if even slightest modifications would be introduced to the processor. Moreover, parallelism is necessary anyways, since already in the near future a processor could have dozens simultaneously working executing units.

Note that our survey is far from being complete, we refer an interested reader to [Int99,Fog00]. However, during finishing our implementations we found that also the official Pentium family optimization manual published by Intel [Int99] is far from being complete. We encountered many problems that could not have been foreseen by using only the official manuals. Often more accurate (although also not complete) information about the Pentium II was found in [Fog00]. In several places of our implementations we performed partial exhaustive search to optimally schedule the instructions. A lot of experience and luck is necessary in optimizing for Pentium II if one desires to avoid exhaustive search himself.

In-Order Decoding. Up to 3 instructions can be decoded to μoperations at time, but only the first decoder can handle instructions with more than one μoperation. It is recommended to order the instructions in the 4-1-1 sequence, which means that only every third instruction could combine in itself of more than one μoperation [Int99]. By this reason, algorithms using only "simple operations" can be potentially implemented faster than those consisting of "complex instructions". However, in some circuimstances it would also beneficial to have at least some complex instructions. Namely, if the code is properly scheduled in a way that exactly (almost) every third instruction has more than one μoperation, the decoder will feed the out-of-order execution pool with pace more than 3 μoperations per cycle. Now, if in some later stage less than 3 μoperations per cycle are fed to the execution unit (say due to the delays in fetching), this unit will not idle waiting for the next instructions from the decoder.

Instruction In-Order Fetching. The Pentium II has 16-byte internal *ifetch buffers* with the peculiarity that a new buffer is forced to start at beginning of an instruction. The first instruction of the ifetch buffer will be always decoded by decoder 0, even if the previous instruction was decoded by the same decoder and hence, other decoders would stay idle. Hence, code reordering and possible use of semantically identical instructions (in general, but not always, *shorter* instructions: for example, `mov eax,[ebx+0]` with `mov eax,[ebx]`) with different length could reduce the number of delays introduced in this stage.

Register In-Order Renaming. Pentium II has 40 hardware registers. The software registers are renamed to hardware registers after a write to (or read from) the software register. After a register has not been used for a while, it automatically retires and the next time the same register is used, a new renaming is performed. It is important to know that *only two register renamings can be done during one machine cycle*. In particular this means that generally it is beneficial to gather all instructions operating on some fixed data chunk together (i.e., to reorder the code in a suitable way). However, it is extremely difficult to detect and remove delays introduced by this stage, and therefore this stage may really become *the* bottleneck in optimization: subtle modification of code may introduce long delays in this stage. We refer to [Fog00] for more information.

Out-of-Order Execution. Pentium II has 5 execution ports (port 0, port 1, $\ldots$, port 4) that can execute instructions out-of-order. Every port has some specific meaning.

Ports 0 and 1 are ALUs (they can perform arithmetic on operands in registers), port 2 performs memory loads. Every memory write counts as two μoperations, one in port 3 (address calculation) and another one in port 4 (memory write). Up to 3 ports can execute an instruction in parallel. There are a number of arithmetic instructions that can only run in port 0 (most importantly, multiplication, rotation and integer register shifts — instructions that are widely used by some AES finalists), while some other instructions (most importantly, MMX register shifts) can only run in port 1. To obtain a throughput near to 3 μoperations per cycle, the instructions should be distributed so that no more than $2/3$ of them are arithmetic, no more than $1/3$ are memory loads and no more than $1/3$ are memory writes: a condition that is very difficult to fulfill in a practical application.

In-Order Retirement After execution, μoperations will retire in-order. During retirement, hardware registers will be written back to software registers and the μoperations leave the instruction pool. Since this is done in-order, several delays can occur, e.g., if speculative out-of-order execution of some earlier long latency instruction is not finished at the moment of retirement.